



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Laboratorio di Sicurezza Informatica

## Binary exploits

**Marco Prandini**

Dipartimento di Informatica – Scienza e Ingegneria

# Attacchi applicativi

- **Gli attacchi applicativi sfruttano le vulnerabilità di:**
  - Sistema Operativo
  - Hardware sottostante
  - Software in esecuzione locale
- **Non sono coinvolti:**
  - Protocolli di rete sotto al livello di applicazione
  - Router e infrastruttura di rete
- **Focus on:**
  - Architettura Intel IA32
  - Sistema Operativo Linux
- **Considerazioni analoghe per altri OS e architetture**



# Obiettivo Exploit

- Lo scopo di un exploit è far eseguire ad un processo operazioni per cui non era stato pensato
- Tre obiettivi (non mutuamente esclusivi):
  - Fermare il processo (Denial Of Service – DOS)
  - Dirottare il flusso di esecuzione (Esecuzione di codice maligno)
  - Ottenere i privilegi di altri utenti
- Le tecniche spaziano su tutti i meccanismi di esecuzione del codice
  - ottimizzazioni hardware (Spectre, Meltdown, Rowhammer, ...)
  - specificità del linguaggio macchina generato da codice compilato
  - allocazione di dati e processi in memoria da parte dei sistemi operativi nelle architetture a microprocessore
  - gestione di input da parte di interpreti di linguaggi di alto livello



# I grandi classici: exploit binari

- La comprensione dei meccanismi alla base delle tecniche di Exploit necessita di una conoscenza basilare di:
  - Disposizione in memoria di un processo
  - Funzionamento dell'architettura del processore su cui lavora il sistema vittima (nel nostro caso IA32)
- Si noti che la maggior parte delle vulnerabilità sono relative a codice scritto in C/C++ e linguaggi derivati. Infatti tali linguaggi, più vicini all'hardware, non realizzano controlli (in automatico) sui dati
- Linguaggi di più alto livello (come ADA, ad es.) in fase di compilazione aggiungono controlli ad ogni istruzione/gruppi di istruzioni



# Processo in memoria

- Nell'OS Linux lo spazio di indirizzamento di un processo in memoria è suddiviso in un insieme di segmenti:
  - Segmento `.text`, che contiene il codice eseguibile
  - Segmento `.data`, contenente i dati inizializzati (variabili statiche inizializzate)
  - Segmento `.bss`, contenente le variabili non inizializzate
  - Stack d'esecuzione, con i record d'attivazione del processo e le variabili locali
  - Heap, segmento di memoria contenete le variabili dinamiche (può crescere dinamicamente)
  - Altri segmenti necessari al funzionamento (`.got`, `.ctor`, `.dtor`)



# Processo in memoria

- Si noti che il programmatore “vede” gli indirizzi virtuali. Questi ultimi sono tradotti dall'OS (+ HW) in indirizzi fisici:

- I segmenti non necessariamente sono contigui

- Andamento indirizzi:

- Lo stack cresce verso il basso
- L'heap cresce verso l'alto

HIGH

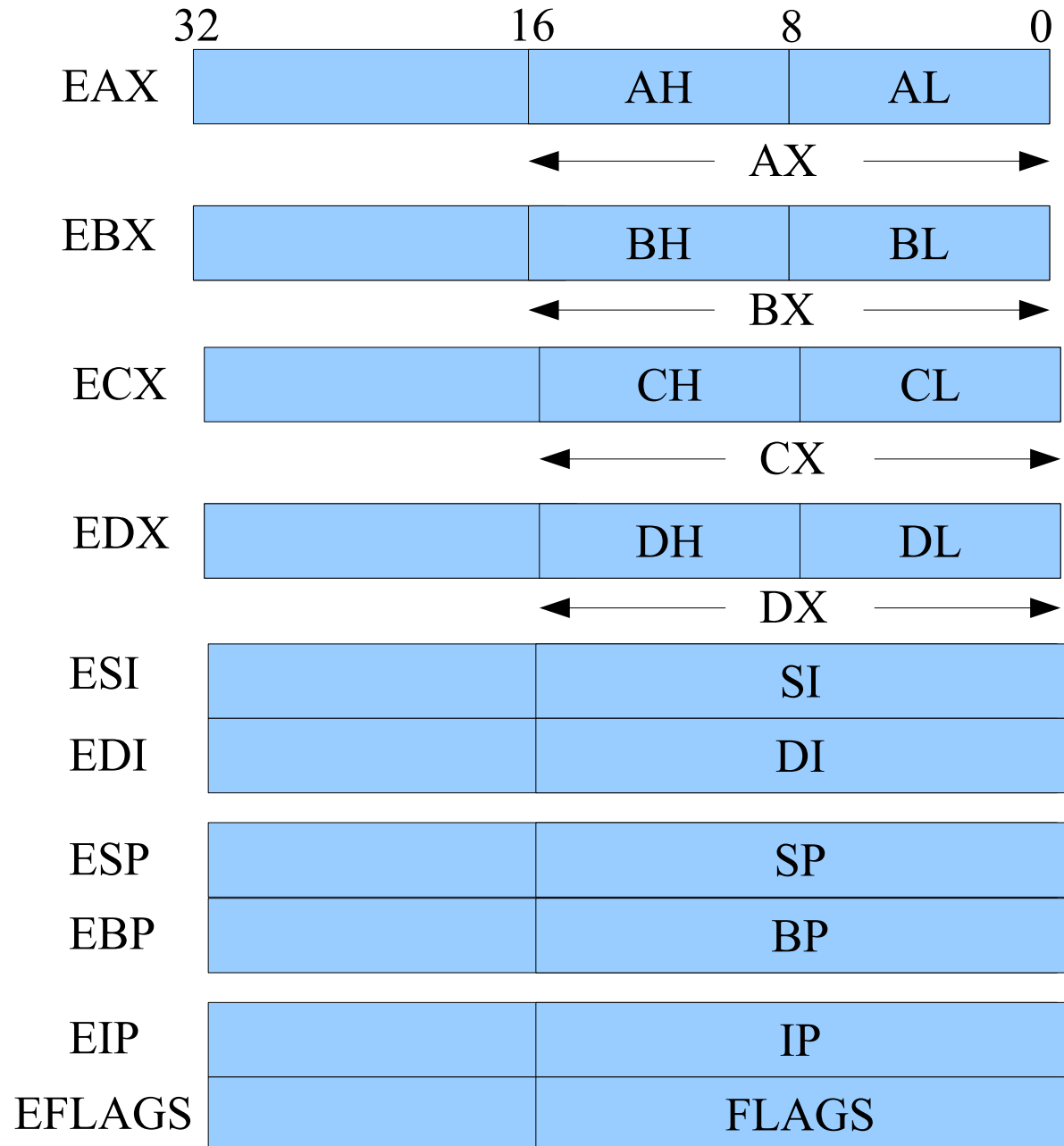


LOW



# Cenni architettura IA32

- L'architettura IA32 è dotata di 4 registri 32 bit “general purpose”:
  - EAX, EBX, ECX, EDX
- Due registri sono usati per le operazioni di copia dati in memoria
  - ESI: source
  - EDI: destination
- Due registri hanno ruoli speciali per il controllo di flusso:
  - EIP: Instruction Ptr
  - EFLAGS: Status register
- Due registri hanno ruoli importanti per la gestione dello stack
  - ESP: stack pointer – punta all’ultima cella occupata dello stack
    - PUSH decrementa ESP di 4 (byte) e scrive il valore sulla cella puntata
    - POP recupera il valore dalla cella puntata da ESP e poi incrementa ESP di 4
  - EBP: base pointer – punta all’inizio dello stack locale; tutte le variabili locali sono referenziate relativamente a EBP



# Convenzioni di chiamata C IA32

- La traduzione C  $\rightarrow$  Assembly adotta convenzioni standard (a differenza di altri linguaggi)
- Convenzione di chiamata di default: **\_\_cdecl**
  - Inserimento sullo stack di tutti i parametri attuali di chiamata in ordine inverso rispetto alla signature del metodo.
  - Chiamata tramite “CALL” salvando l'indirizzo di ritorno sullo stack.
  - EBP contiene il riferimento per le variabili locali al chiamante, non deve essere perso, ma il chiamato ha bisogno di settarlo al proprio “sistema di riferimento”
    - Il chiamato salva il contenuto del registro EBP ponendolo sullo stack e aggiorna il valore di EBP al contenuto attuale di ESP
  - Il chiamato ritorna il risultato delle proprie computazioni nel registro EAX.
  - E' compito del chiamato ripristinare il valore originario di EBP, con quello (da lui) salvato sullo stack in precedenza.
  - E' compito del chiamante rimuovere i parametri attuali dallo stack



# Convenzioni di chiamata C IA32

## ■ Altre convenzioni

### — `__stdcall`

- L'unica differenza con la modalità precedente è che è responsabilità del chiamato eliminare i parametri dallo stack.

### — `__fastcall`

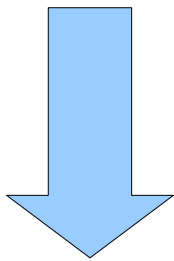
- La differenza con la `__cdecl` è che i parametri attuali non sono passati via stack bensì tramite i registri generali (da EAX a EDX e ESI e EDI, quindi con un limite di 6 parametri di 32 bit)



# Esempio di chiamata

Codice chiamante (convenzione `__cdecl`)

```
...  
int result;  
...  
result = sum(4, 5);  
...
```



Compilazione

```
0x80401000 result: DW 1  
...  
0x8040200A PUSH 0x5  
0x8040200F PUSH 0x4  
0x80402015 CALL _sum  
0x8040201A ADD ESP, 0x8  
0x8040201F MOV result, EAX  
...
```

Indirizzo di ritorno

Il chiamante immette i parametri della funzione in cima allo stack in ordine inverso

Il chiamante invoca la funzione tramite l'operazione CALL = PUSH dell'indirizzo di ritorno + caricamento indirizzo funzione in EIP

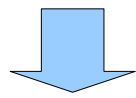
Di ritorno dalla funzione, il chiamante rimuove dallo stack i parametri passati

Il chiamante prende il risultato dal registro EAX

# Esempio di chiamata

Codice chiamato (convenzione \_\_cdecl)

```
int __cdecl sum(int a, int b) {  
    int c;  
    c = a+b;  
    return c;  
}
```



Compilazione

```
PUSH    EBP  
MOV     EBP, ESP  
SUB     ESP, 0X4  
MOV     [EBP-4], [EBP+8]  
ADD     [EBP-4], [EBP+12]  
MOV     EAX, [EBP-4]  
MOV     ESP, EBP  
POP     EBP  
RET
```

Salva il contenuto di EBP, e  
memorizza in EBP il puntatore al  
frame pointer (contenuto di ESP)

Riserva sullo stack lo spazio per  
la variabile locale c

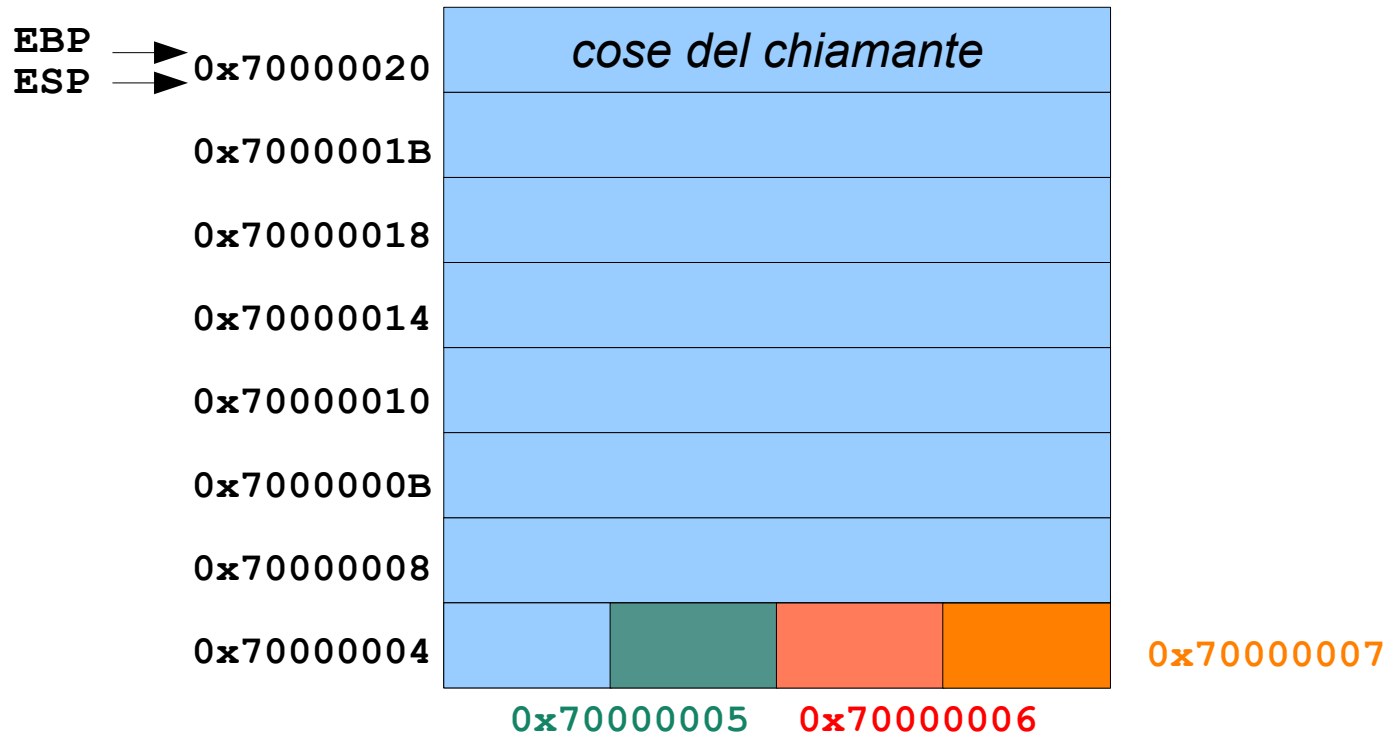
Effettua le operazioni usando come  
Riferimento (in base a cui muoversi  
sullo stack) il registro EBP

Salva il risultato in EAX

Ripristino di EBP e ESP

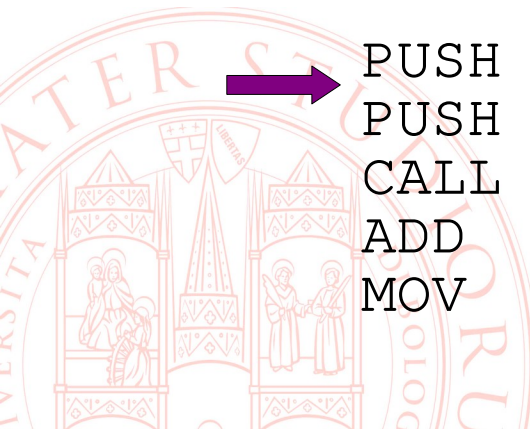
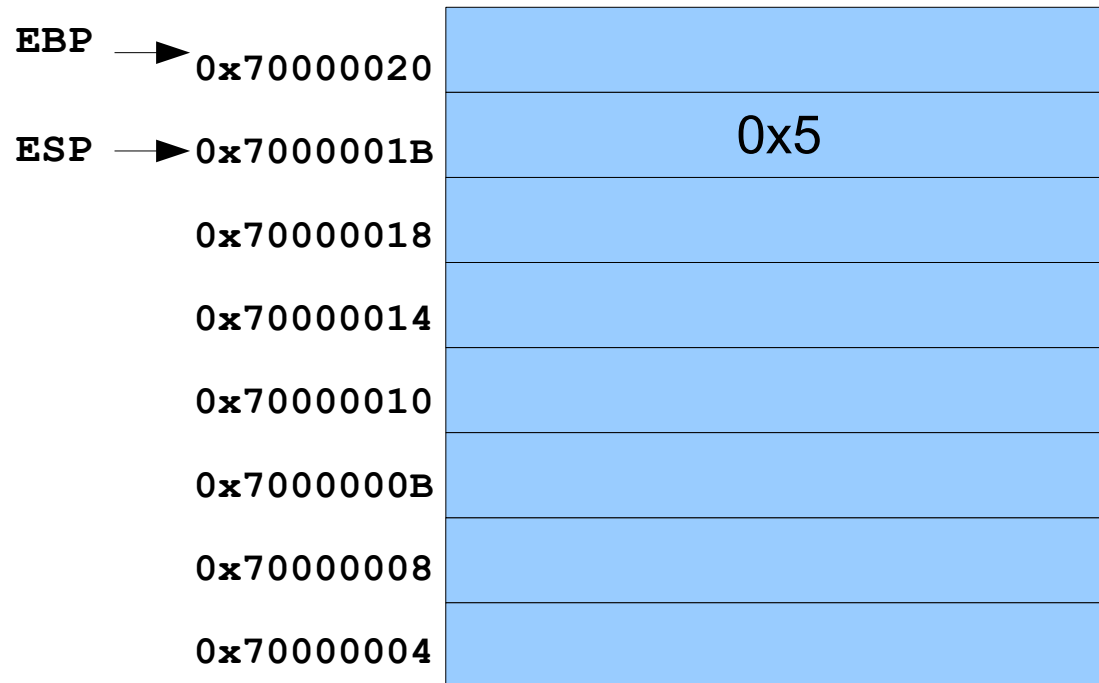
Ritorno al chiamante =  
POP dell'indirizzo in EIP


# Evoluzione dello stack



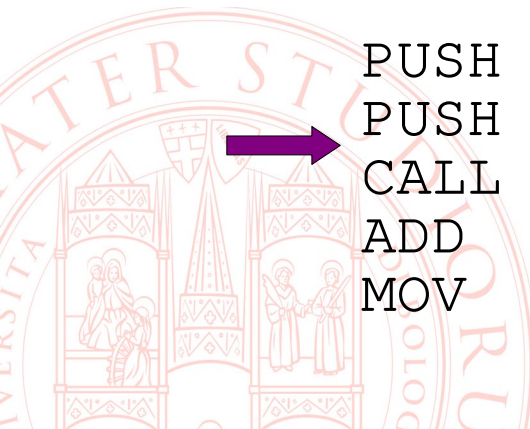
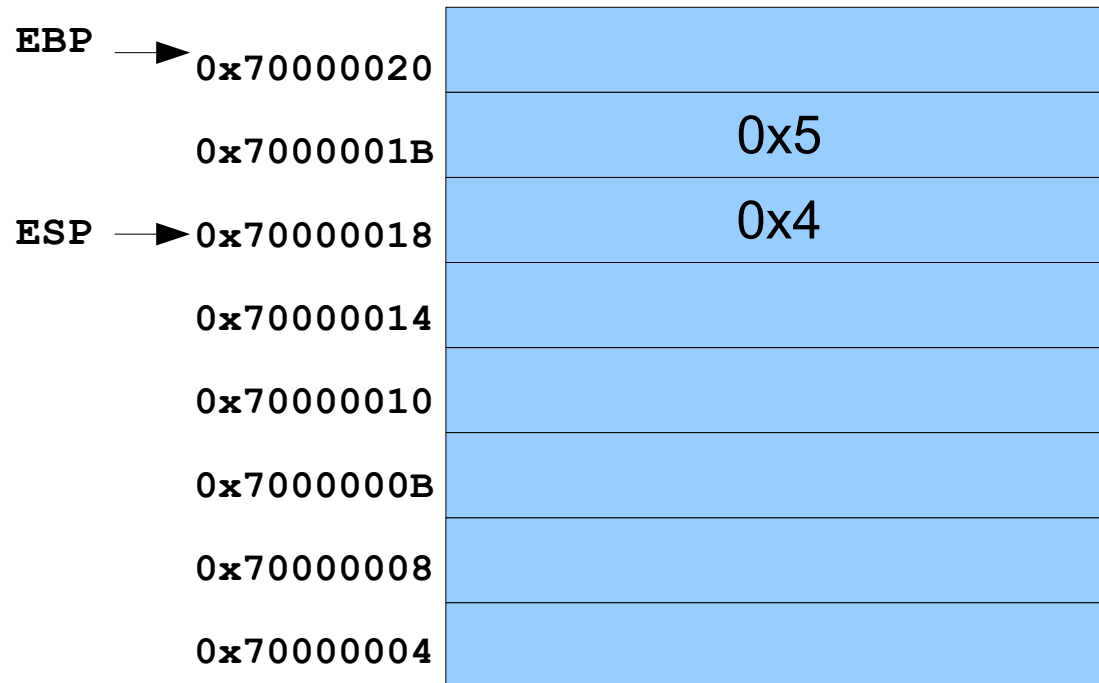
```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

# Evoluzione dello stack



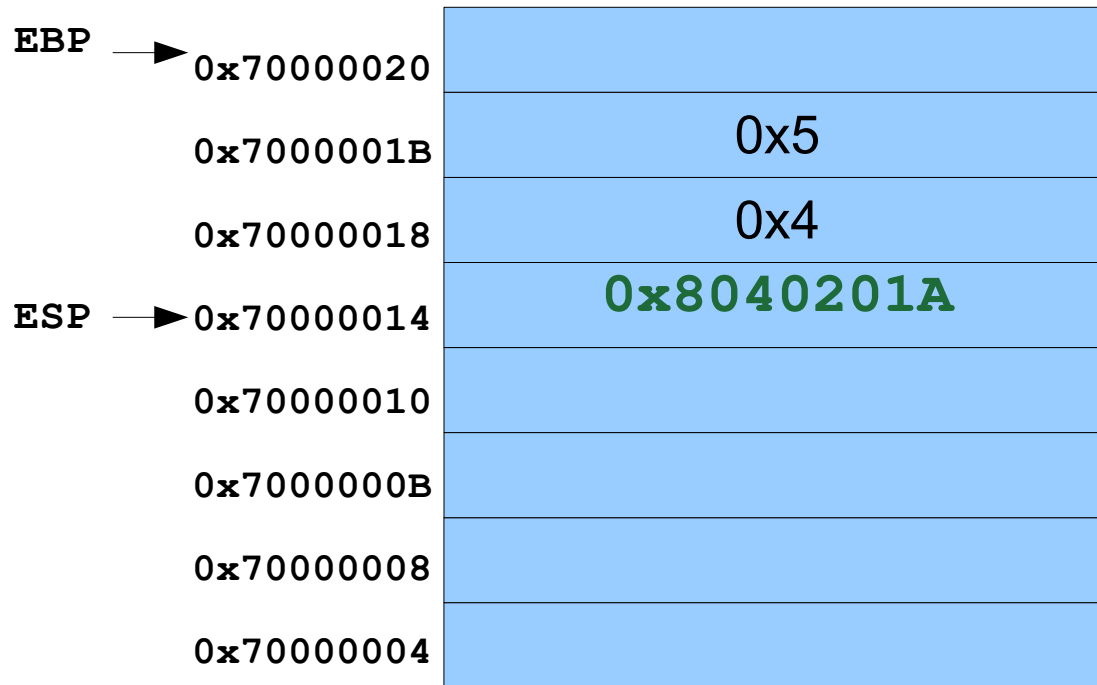
 PUSH 0x5  
PUSH 0x4  
CALL \_sum  
ADD ESP, 0x8  
MOV result, EAX

# Evoluzione dello stack




```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

# Evoluzione dello stack

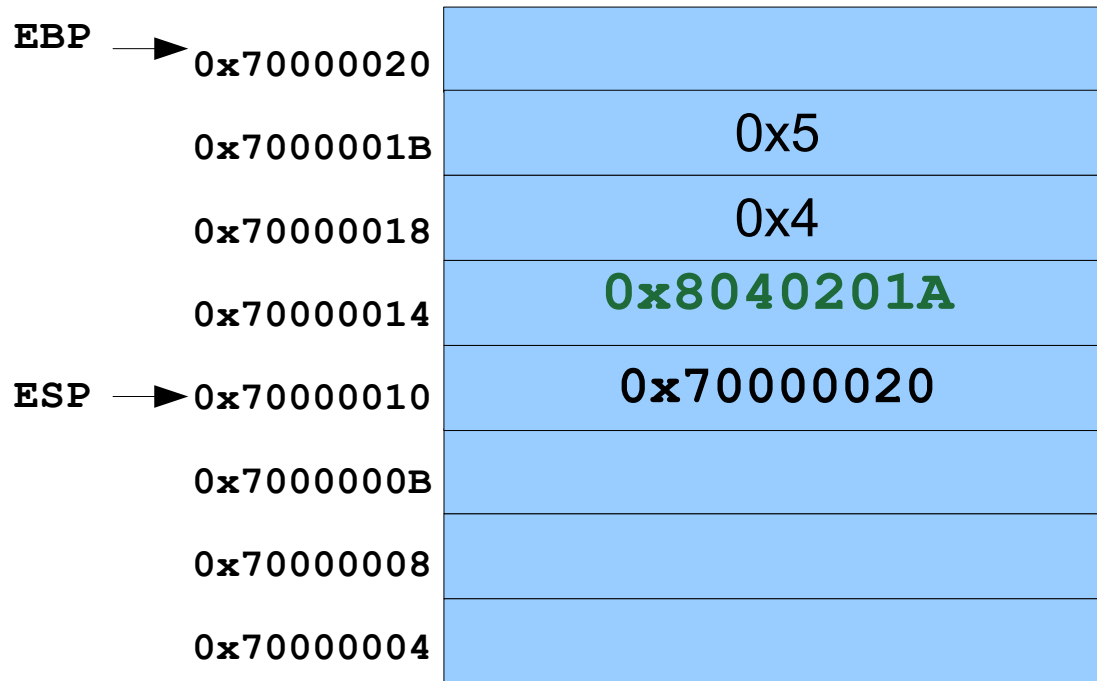


```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```




```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

# Evoluzione dello stack



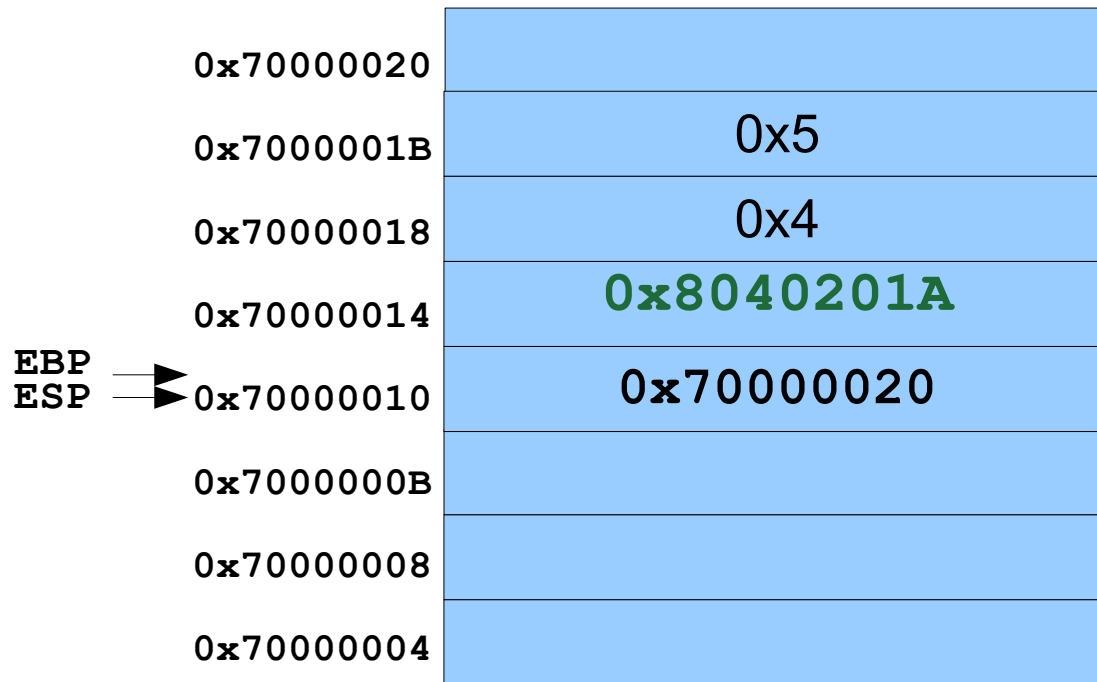
```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```



```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```



# Evoluzione dello stack



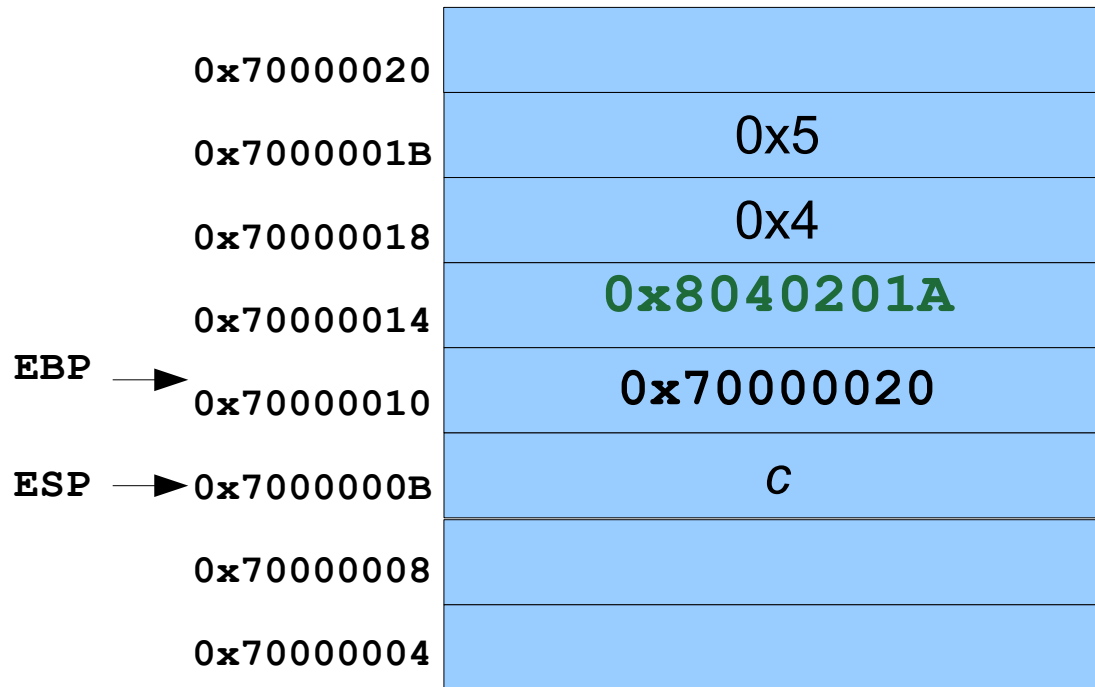
```

PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
    
```

```

PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
    
```

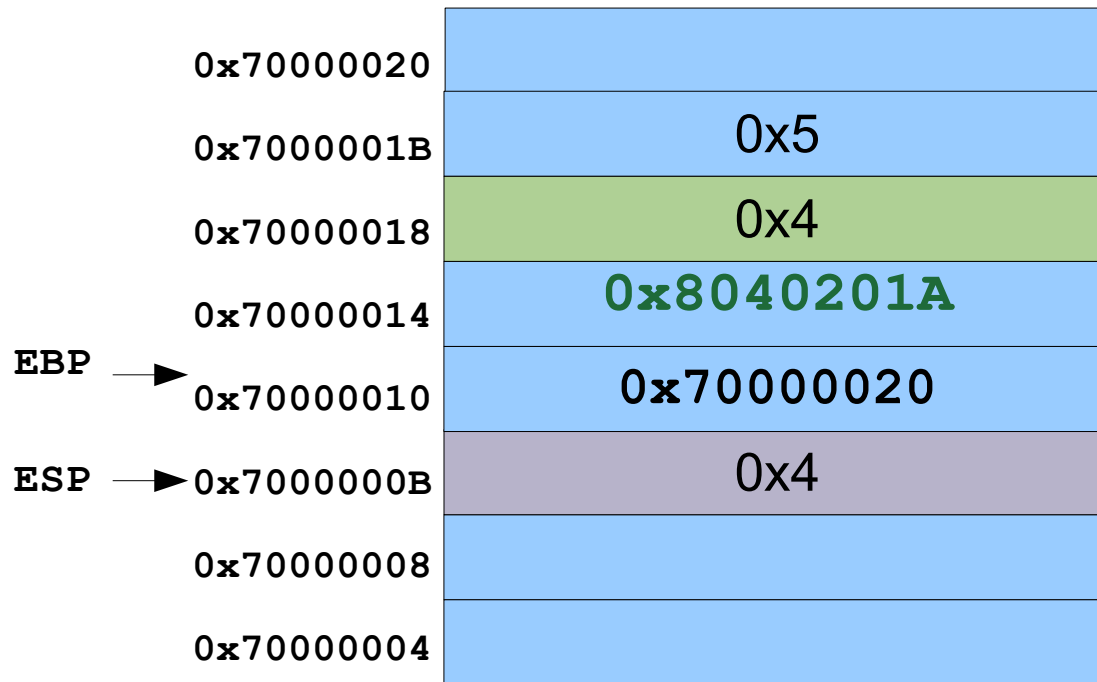
# Evoluzione dello stack



```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

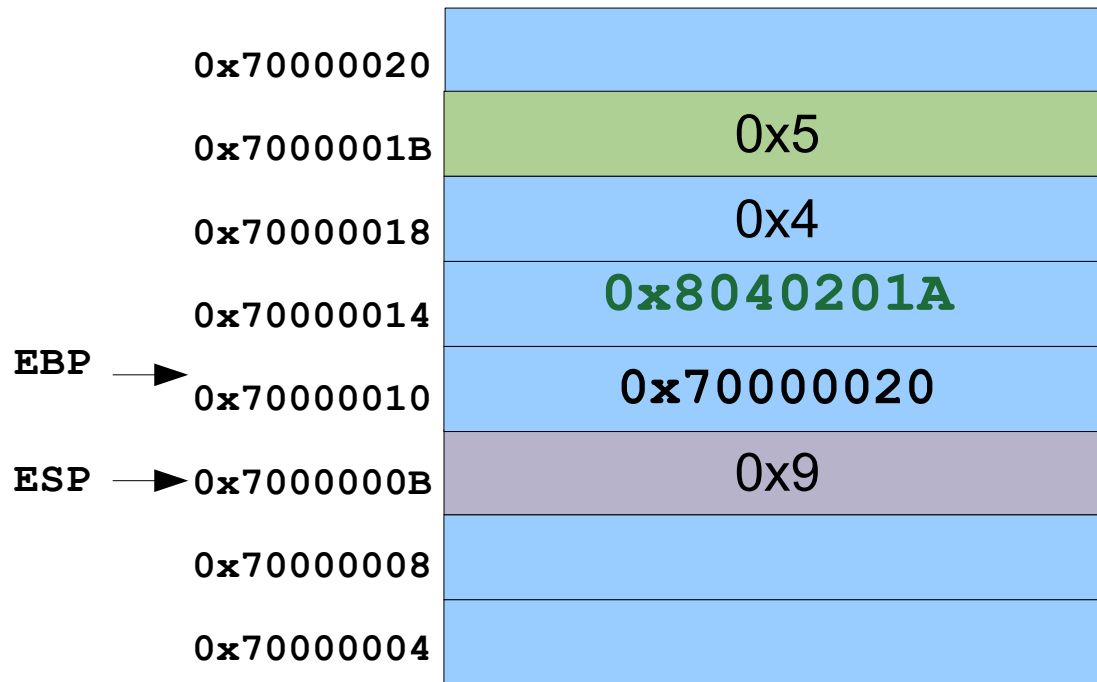
# Evoluzione dello stack



```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

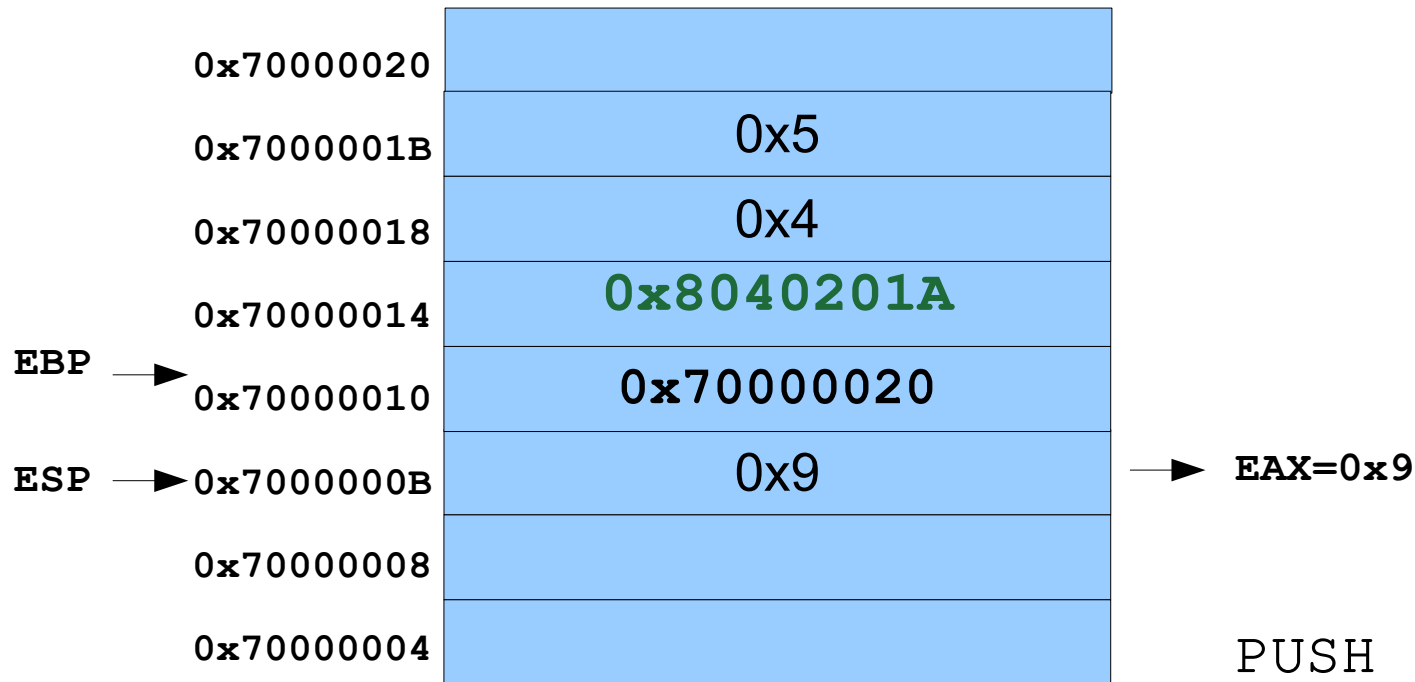
# Evoluzione dello stack



```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

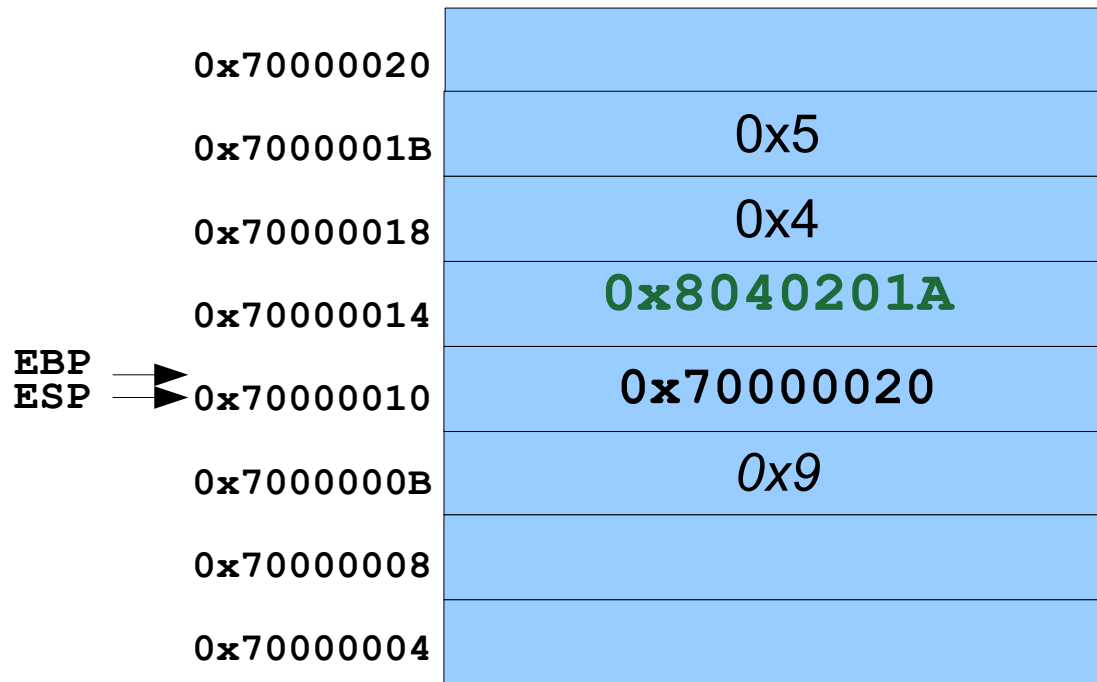
# Evoluzione dello stack



```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

# Evoluzione dello stack



EAX=0x9

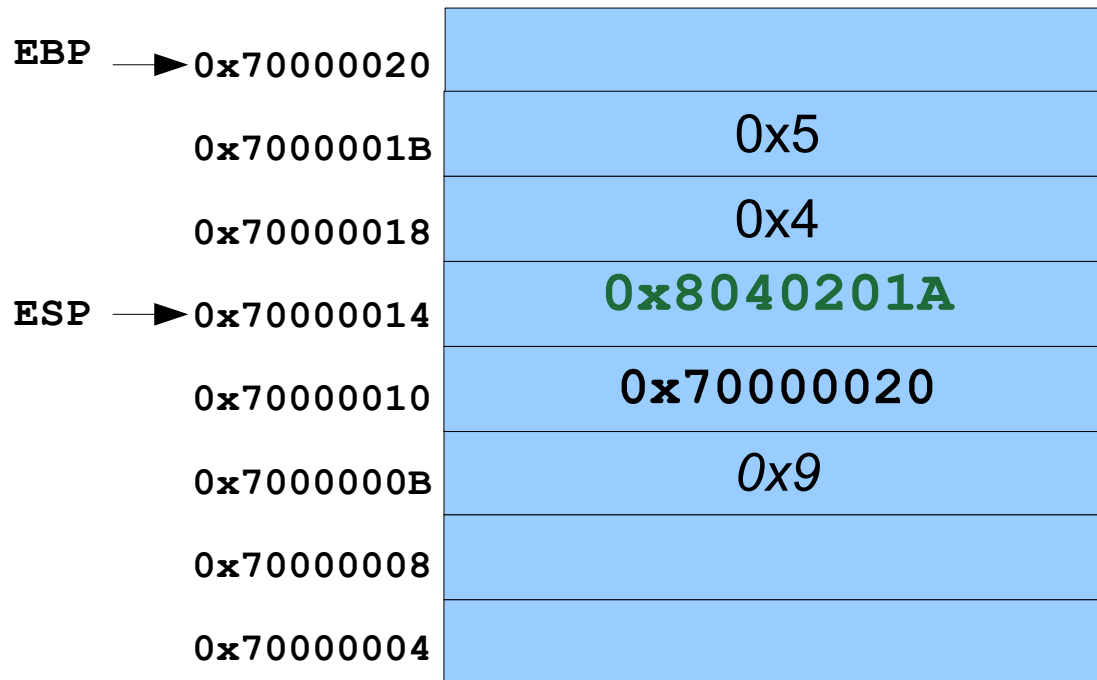
```

PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
    
```

```

PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
    
```

# Evoluzione dello stack

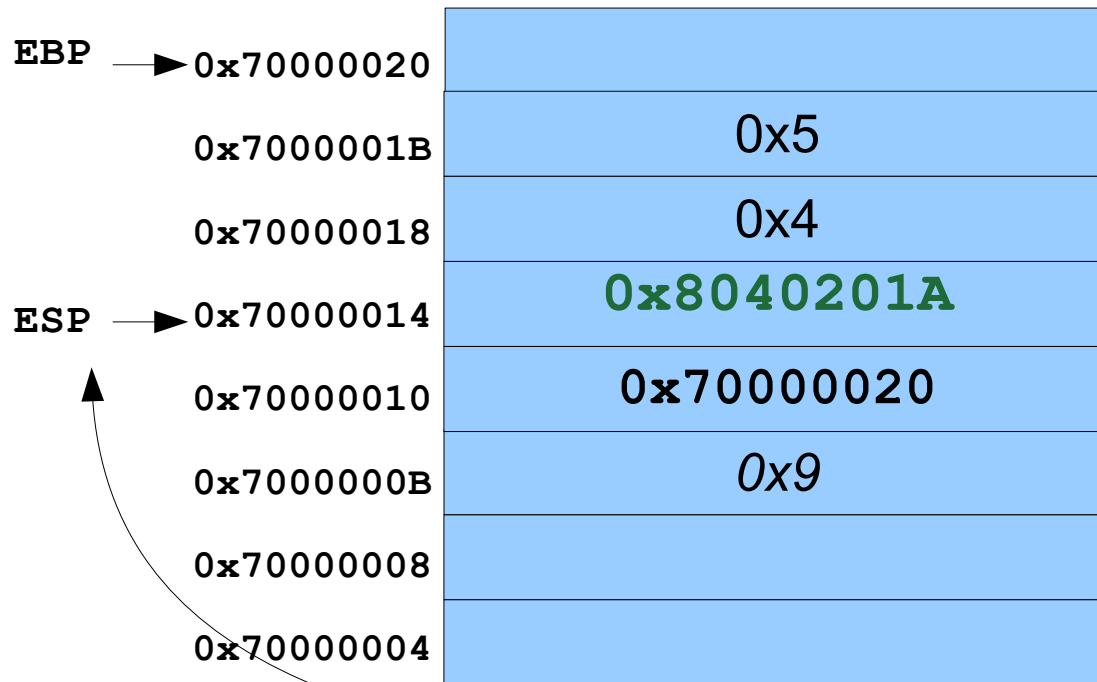


EAX=0x9

```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

# Evoluzione dello stack



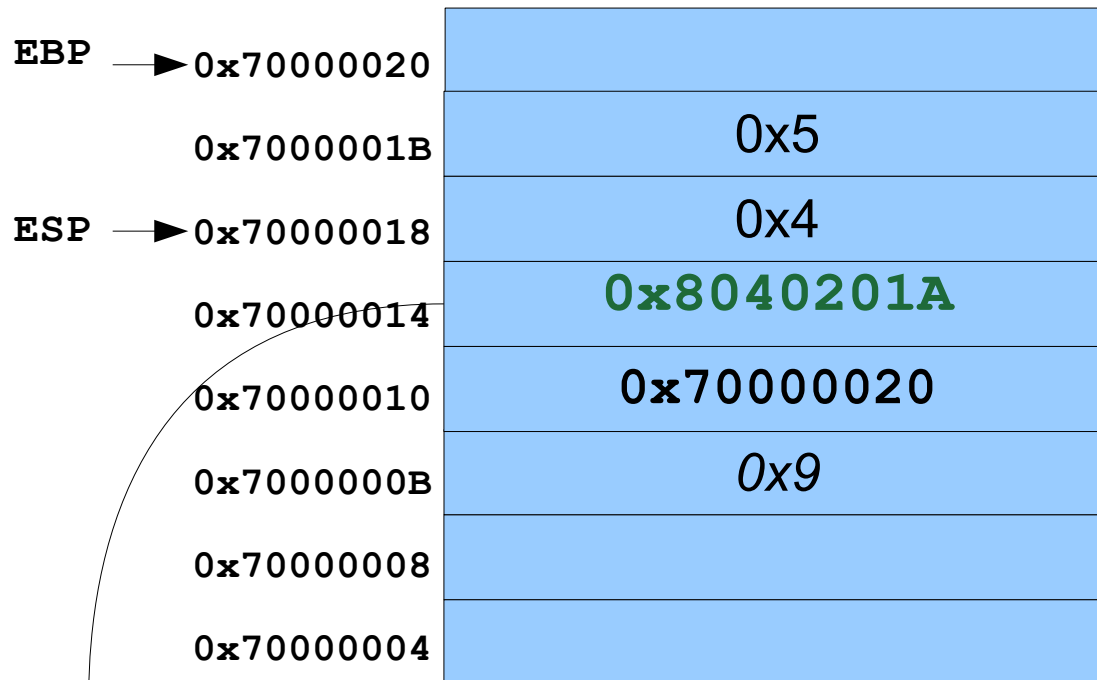
EAX=0x9

```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```



# Evoluzione dello stack

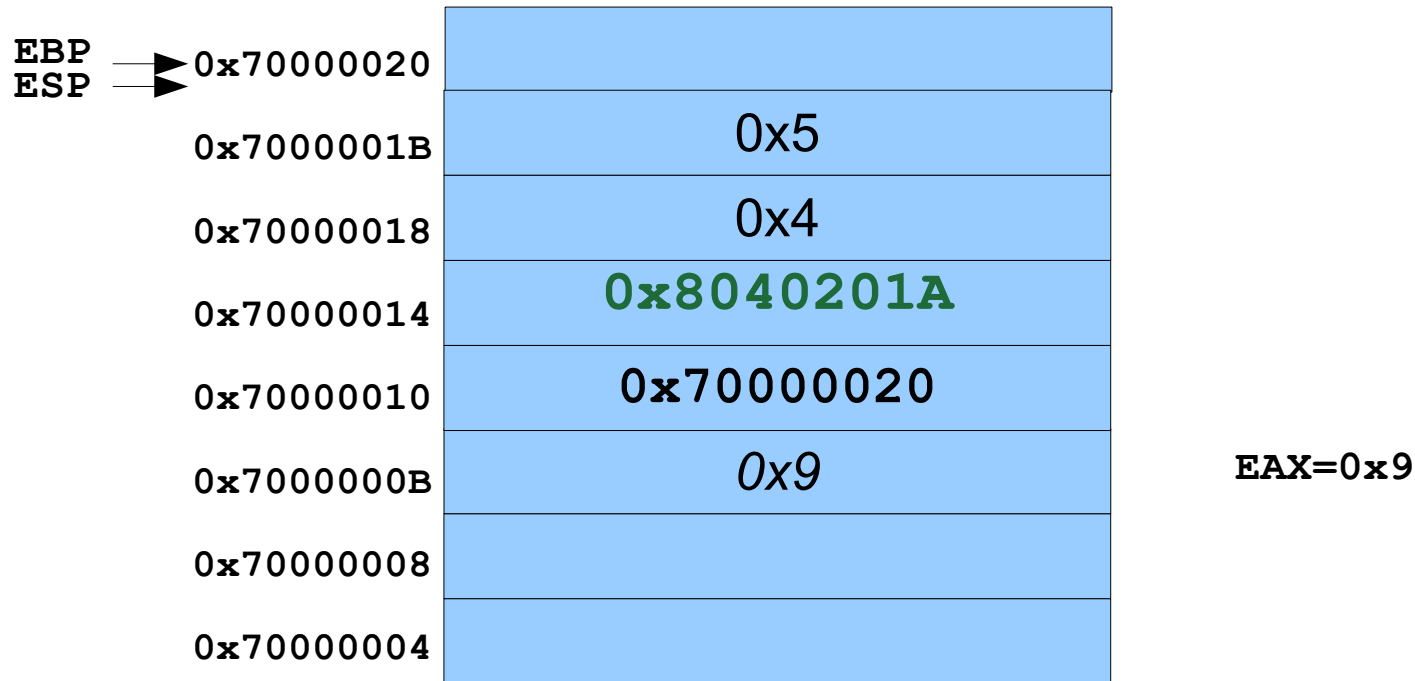


```
PUSH    0x5
PUSH    0x4
CALL    _sum
ADD     ESP, 0x8
MOV     result, EAX
```

EAX=0x9

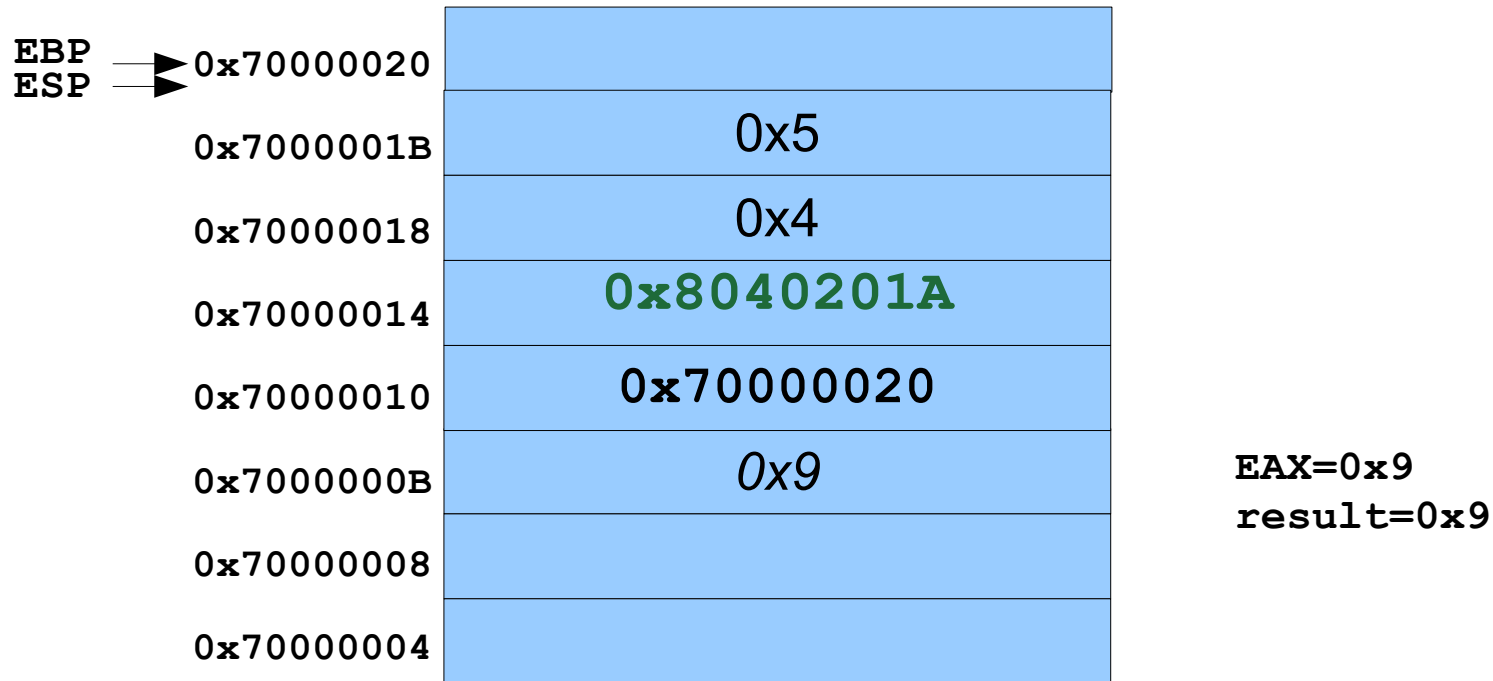
```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

# Evoluzione dello stack



PUSH 0x5  
PUSH 0x4  
CALL \_sum  
ADD ESP, 0x8  
MOV result, EAX

# Evoluzione dello stack



PUSH 0x5  
PUSH 0x4  
CALL \_sum  
ADD ESP, 0x8  
MOV result, EAX

# Stack Overflow (1/9)

## ■ Prerequisiti per l'attuazione:

- Presenza di un buffer locale ad una funzione (tale buffer si troverà quindi sullo stack).
- Possibilità di alimentare il buffer con input esterni (ad es. da tastiera, da file, da rete).

## ■ Modalità di attuazione:

- L'attaccante riempie il buffer con dati fittizi (padding), sforandone i limiti, fino ad arrivare a porre sullo stack “nella posizione giusta” un nuovo indirizzo di ritorno dalla chiamata (sovrascrivendo quello preesistente).

## ■ Risultato:

- Il flusso di controllo non procede con il ritorno al chiamante “lecito” bensì al “nuovo” indirizzo di ritorno posto dall'attaccante.



# Stack Overflow (2/9)

- Un buffer locale ad una funzione/routine è realizzato con un blocco di memoria riservato sullo stack (di lunghezza finita e predeterminata).
- Il linguaggio C non controlla che i dati con cui alimentare i buffer/array abbiano una lunghezza congrua (tale controllo è a carico del programmatore).
- Si ricordi che:
  - Lo stack cresce con indirizzi decrescenti (cioè i dati “più vecchi” hanno indirizzi più alti)
  - Il riempimento del buffer avviene invece per indirizzi crescenti (sebbene questo si trovi sullo stack)



# Stack Overflow (3/9)

## ■ Conseguenza della strategia di riempimento dello stack:

- In assenza di controlli di overflow, i dati in eccesso (immessi nel buffer) vanno a sovrascrivere i dati dello stack immessi in precedenza.

## ■ Esempio

```
void function() {  
    char buffer[10];  
    gets(buffer);  
    ...  
}
```

- La funzione gets legge una stringa dallo stdin ma non effettua controlli sulla lunghezza.



# Stack Overflow (4/9)

■ Esempio di disposizione in memoria.

■ L'attaccante scrive una stringa lunga più del dovuto

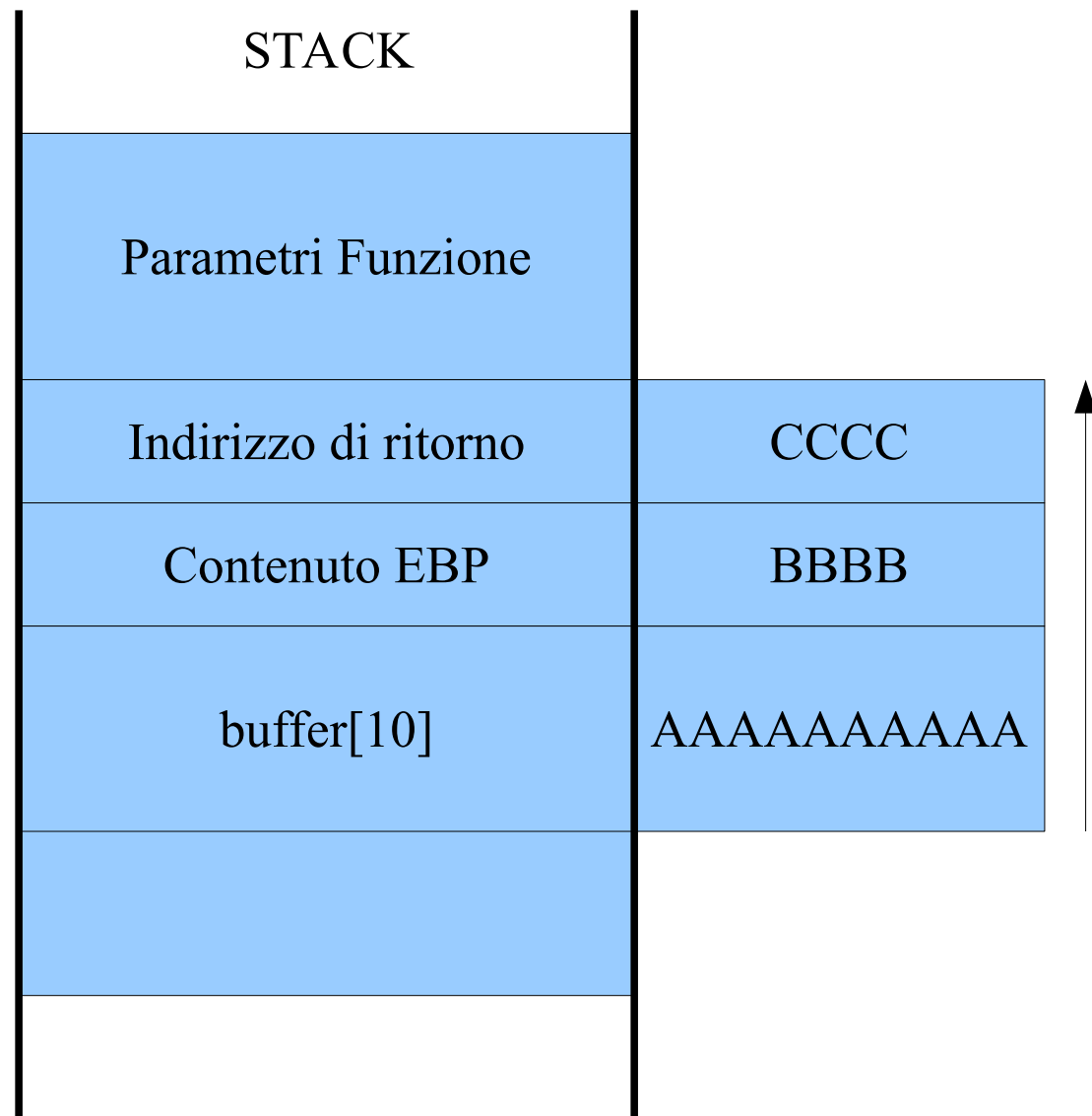
■ Nell'esempio di prima:

- 10 Byte di padding
- 4 byte per coprire EBP
- 4 byte per sovrascrivere l'indirizzo di ritorno

■ Si noti come l'overflow sovrascriva l'indirizzo di ritorno.

0x128

0x90



Stringa: “AAAAAAAAAAAABBBBCCCC”

# Stack Overflow (5/9)

## ■ Conseguenze dell'attacco:

### – Denial Of Service

- Se l'indirizzo di ritorno “illecito” è relativo ad una posizione in memoria non accessibile (dal programma in esecuzione) avviene un crash per “segmentation fault”.

### – Controllo del flusso. L'attaccante prende il controllo dell'esecuzione. Si hanno due alternative:

- Shell Coding: si “inietta” (come parte della stringa) un pezzo di codice maligno preparato in precedenza. In questo modo, è possibile eseguire codice con i privilegi del processo vittima.
- Return to LibC: la stringa iniettata per overflow scrive sullo stack i dati necessari per effettuare una invocazione di una funzione di libreria C.





# Stack Overflow (6/9)

- Un semplice esempio di controllo del flusso
- In questo programma l'input copiato in un buffer determina se un utente ha i diritti di eseguire una porzione di codice o meno (ad es. potrebbe essere richiesta la password da linea di comando)

```
int validate() {
    char buffer[10];
    gets(buffer);
    if(//Valid input)
        return 1;
    else
        return 0;
}

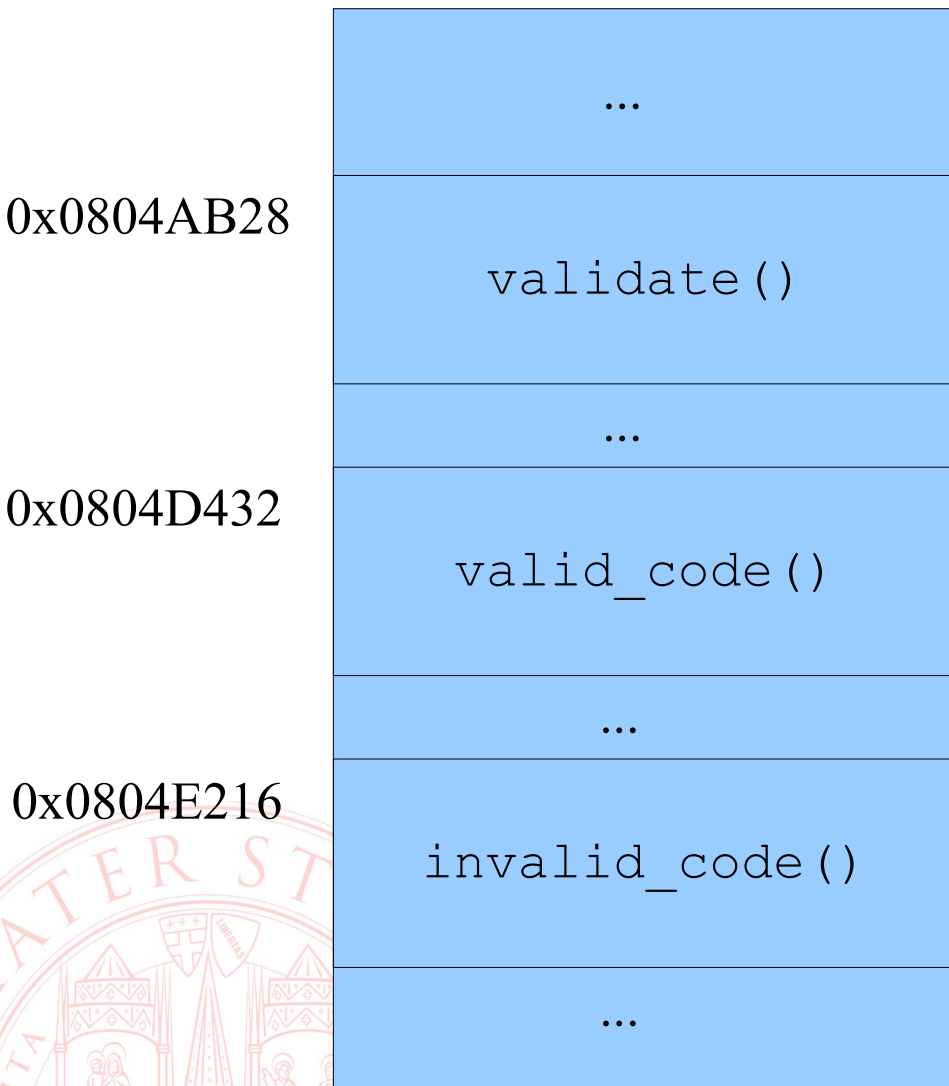
void valid_code() {
    // Codice per utenti autorizzati
    ...
}

void invalid_code() {
    // Codice per utenti non aut.
    ...
}
```



# Stack Overflow (7/9)

Segmento .text



- L'obiettivo di un attaccante, che non ha le credenziali adatte a fare in modo che `validate` ritorni 1, è di forzare l'esecuzione del codice di `valid_code`
- `validate` utilizza la funzione `gets` per ottenere la password
- La `gets` è non sicura
- In tali condizioni un attacco di stack overflow è molto semplice

# Stack Overflow (8/9)

- Nel momento in cui `validate` richiede l'input all'utente l'attaccante prepara una stringa così composta

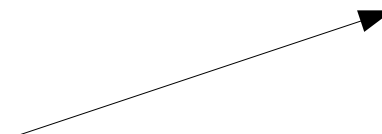
- 10 byte di padding
- 4 byte per “scavalcare” l'EBP
- 4 byte per scrivere l'indirizzo di `valid_code`: **0x0804D432**

- Quando `validate` eseguirà la `RET`, invece che tornare al chiamante, salterà a `valid_code`

E' necessario tradurre l'indirizzo di `valid_code` nella sua rappresentazione ASCII. Il risultato comprende anche caratteri non stampabili. Esistono varie tecniche per passare al processo tali caratteri (ad es. `printf` di bash).

Stringa d'attacco: **“AAAAAAAAAABBBB\x32\xD4\x04\x08”**

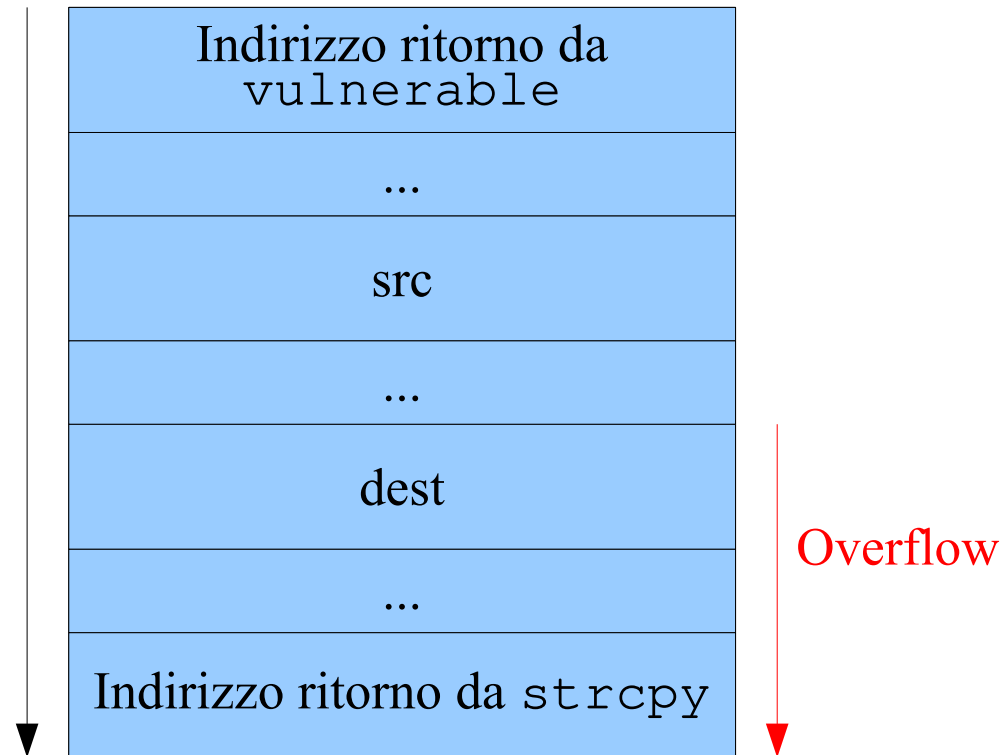
L'indirizzo è fornito con i byte in ordine inverso:  
l'architettura IA32 è infatti **little endian**



# Stack Overflow (9/9)

- L'attacco di Stack Overflow può essere utilizzato anche su architetture che presentano stack con indirizzi crescenti

```
void vulnerable() {  
    ...  
    char src[10];  
    char dest[10];  
    ...  
    gets(src);  
    ...  
    strcpy(dest, src);  
    ...  
}
```



- In questo caso, inserendo una stringa più lunga del dovuto in `src`, nel momento in cui verrà richiamata la `strcpy` per copiarla in `dest`, essa andrà a sovrascrivere l'indirizzo di ritorno della `strcpy` stessa e non di `vulnerable`

# Stack Overflow – Canarini (1/3)

## ■ Canarini

- Prima forma di contromisura
  - Deve il suo nome alla similitudine storica con i canarini dei minatori in grado di rilevare fughe di gas
- Il concetto alla base è di porre sullo stack, prima di un buffer, un dato di riferimento.
- Il processo è in grado di rilevare un tentativo di attacco verificando l'integrità di tale dato.
  - In caso di attacco con overflow il dato viene sovrascritto e la verifica da esito negativo
  - Tale protezione è realizzata tramite la collaborazione congiunta di compilatore e libreria standard
  - Non è un meccanismo fornito dall'OS o dall'HW
  - Generazione del dato e verifica causano overhead

# Stack Overflow – Canarini (2/3)

## ■ Funzionamento in GCC

- Tale protezione era inizialmente fornita come patch ProPolice (a partire dalla versione 3.x)
- Inclusa nel main branch a partire dalla versione 4.1

## ■ Abilitazione:

- Non è abilitato di default su tutti gli OS / Distro Linux
- `-fstack-protector` **abilita solo per buffer di stringhe**
- `-fstack-protector-all` **abilita per tutti i tipi di buffer**
- `--param ssp-buffer-size=` **imposta una soglia di dimensione del buffer oltre la quale la protezione viene attivata. Questo evita che la protezione venga attivata per tutte le chiamate a funzione, riducendo l'overhead.**



# Stack Overflow – Canarini (3/3)



- Ad ogni chiamata di funzione:
  - Si genera e scrive un valore casuale di 4 byte
- L'attaccante dovrebbe sovrascrivere anche il canarino per modificare l'indirizzo di ritorno (non può “scavalcarlo”)
- La modifica del canarino viene rilevata nel momento in cui si ritorna dalla chiamata
- L'azione di default in tal caso è la terminazione del processo
- Tale evento può essere inoltre catturato tramite segnali dell'OS
- Attaccabile a forza bruta o con tecniche più sofisticate



# Shellcoding (1/6)

- Non è una forma di attacco alternativa, bensì è un tecnica per sfruttare lo stack overflow.
- Consiste nell'iniettare (tramite stack overflow):
  - Codice maligno
  - Indirizzo di ritorno che punti al codice di cui sopra
- L'obiettivo tipico di tale approccio è l'apertura di una shell (da cui il nome), con i privilegi del processo attaccato (tipicamente root o con il bit setuid attivato).
- Il principio alla base di tale tecnica è utilizzabile anche con altri tipi di attacco alternativi allo stack overflow.





# Shellcoding (2/6)

- Un esempio di codice maligno che realizza l'invocazione della system call exit con il valore 0, terminando il processo è il seguente:

```
mov EBX, 0  
mov EAX, 1  
int 0x80
```

I parametri di una system call devono essere caricati nei registri in ordine EBX, ECX, EDX, ESI, EDI. Nel nostro caso l'unico parametro è il valore 0.


Il registro EAX va predisposto con l'identificativo numerico della system call. nel nostro caso exit=1

Si genera un interrupt software 0x80  
(che corrisponde al gestore delle system call in Linux)

# Shellcoding (3/6)


- Quando l'iniezione del codice avviene con lettura di stringa, non tutti i byte sono ammissibili

- Tornando all'esempio di prima:

<code>mov EBX, 0</code>		BB	00	00	00	00
<code>mov EAX, 1</code>		B8	01	00	00	00
<code>int 0x80</code>		CD	80			

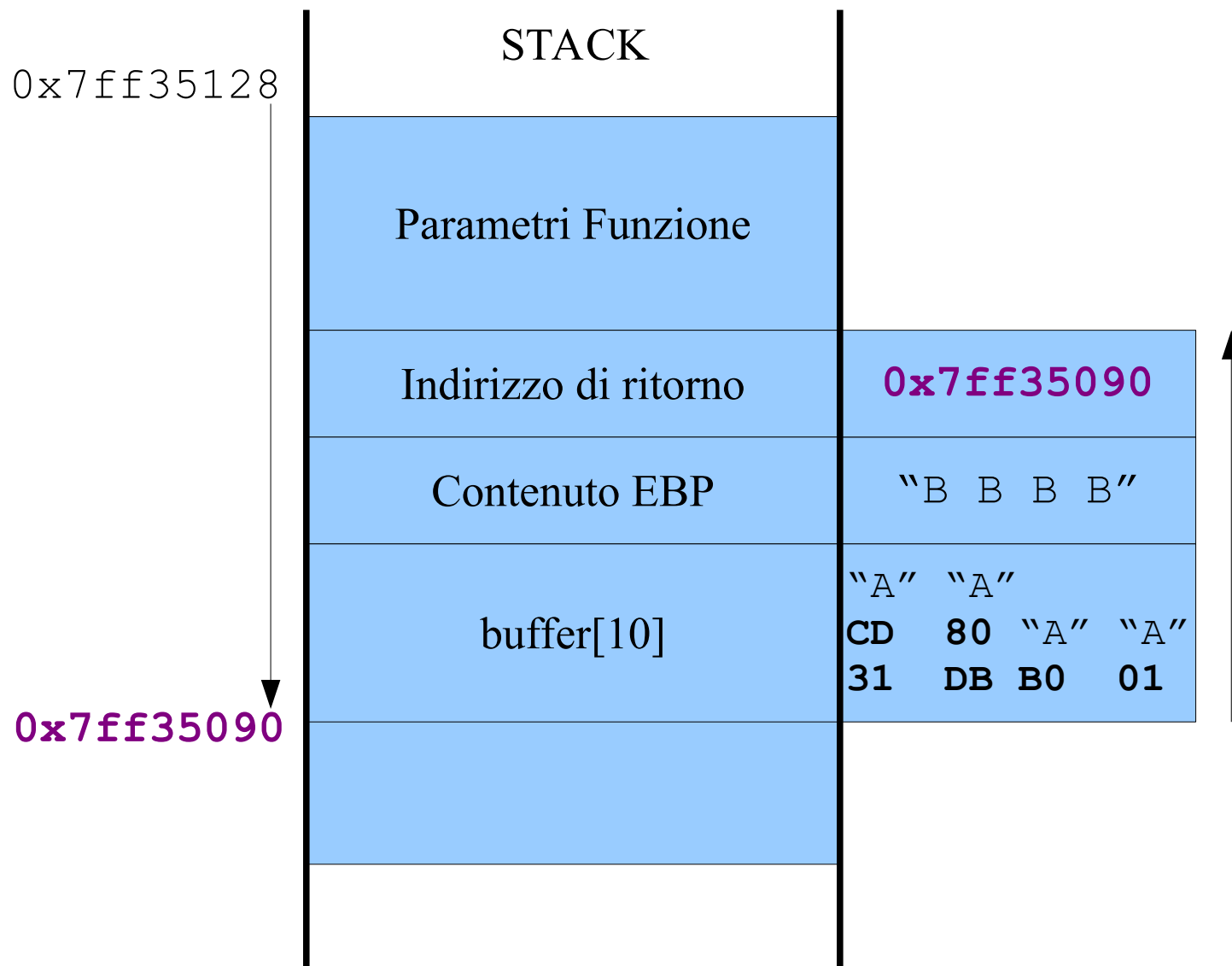
- Nel risultato esadecimale c'è una serie di byte uguali a 0
- Il primo di questi verrà interpretato dalle routine di lettura stringhe come **terminatore di stringa**
- Uno shellcode, per poter essere iniettabile, deve essere sottoposto ad un'operazione di **Zeros Cut-off**
- Nell'esempio si può agire in questo modo:

NB: AL è l'insieme di  
8 bit meno significativi  
Del registro EAX

<code>xor EBX, EBX</code>		31	DB
<code>mov AL, 1</code>		B0	01
<code>int 0x80</code>		CD	80

# Shellcoding (4/6)

- Dopo aver preparato il codice assembly da iniettare si procede con il buffer overflow
- Si vuole porre l'indirizzo di ritorno al punto di partenza del codice iniettato
- Il problema che si presenta è come stabilire l'indirizzo!



Stringa: "\x31\xDB\xB0\xCD\x80AAAABBBB\x90\x50\xF3\x7F"

# Shellcoding (5/6)

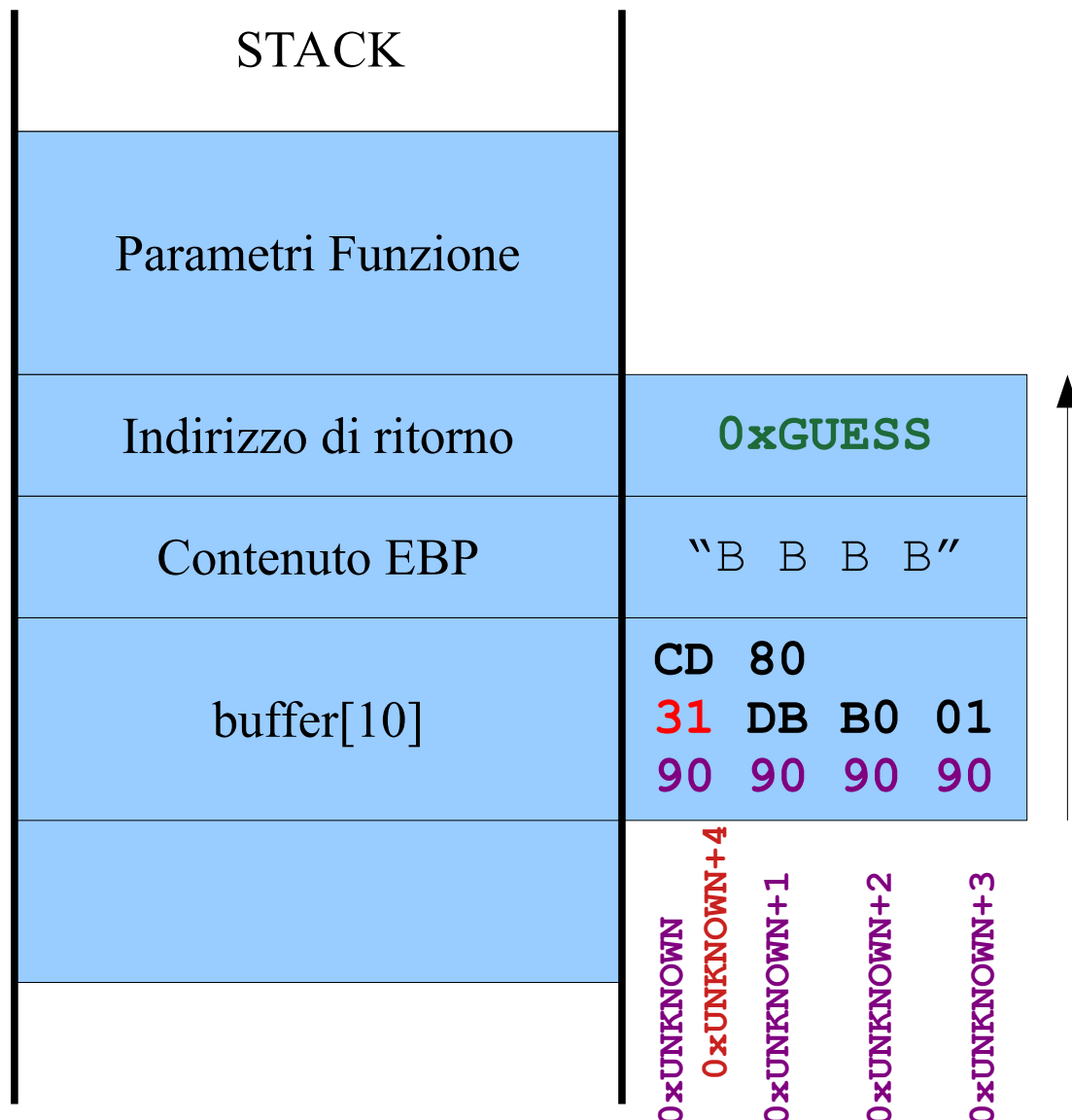
- Per calcolare/intuire l'indirizzo di ritorno esistono varie tecniche:
  - Tipicamente il segmento `.stack` di un processo comincia sempre a partire dallo stesso indirizzo
  - L'attaccante può procurarsi il codice sorgente del programma da attaccare e la sua immagine binaria (facile se la vittima usa una qualsiasi distro Linux)
- L'evoluzione dello stack però dipende dalla specifica esecuzione del processo!
- Per avere maggiori garanzie sulla riuscita si può ricorrere ad un “NOP Sled” (slitta di NOP)



# Shellcoding (6/6)

- Il NOP Sled è una sequenza di NOP (operazione assembly che non effettua nulla) che precede lo Shellcode.
- E' sufficiente che il RET Address cada in un punto qualsiasi del NOP Sled

0xUNKNOWN



Stringa: "\x90\x90\x90\x90\x90\x31\xDB\xB0\xCD\x80BBBB\xGUESS"

$UNKNOWN \leq GUESS \leq UNKNOWN+4$

in più potrei sfruttare i 4 byte di EBP

# NX Stacks

- Una seconda contromisura alla possibilità di eseguire codice maligno dallo stack è fornita dagli **Stack Non Eseguibili**
- E' una feature fornita dall'HW ma che deve essere supportata dall'OS
- Alcune architetture permettono di impostare un flag associato a pagine di memoria
  - Tale flag, che deve essere impostato dall'OS, indica se la pagina contiene codice che può essere eseguito o meno
  - Intel commercializza questa feature con il nome di XD bit (eXecution Disable bit) ed è stata introdotta solo a partire dai processori a 64 bit
  - Esistono alcuni OS (Solaris) che implementano tale feature in software, ma ciò causa un leggero overhead



# NX Stacks

- Tale feature non è presente sui processori INTEL/AMD a 32 bit
- Linux supporta gli stack non eseguibili a partire dal kernel 2.6.8 per architettura x86\_64
  - Introdotta con la patch PAX/grsecurity
  - La versione x86 a 32 bit può sfruttare tale feature solo se in esecuzione su un processore x86\_64
- Un'evoluzione di tale tecnologia è la **W<sup>X</sup>**
  - Piuttosto che marcare una pagina come non eseguibile, ogni pagina viene marcata come Eseguibile (eXecutable) o Scrivibile (Writable) ma non entrambe
  - Offre una sicurezza maggiore rispetto al bit XD, che comunque permette di eseguire codice da pagine scrivibili
  - In Linux è implementato parzialmente da ExecShield





# Anti NX-stack?

## ■ RET2LIBC / RET2SYSCALL

- Tramite Stack Overflow si dirotta il flusso di esecuzione, piuttosto che verso codice sullo stack protetto da NX, verso una (o più) funzioni della onnipresente libreria C, oppure verso una system-call del s. o.

## ■ Format Strings

- Tramite la stringa di formato passata a printf si inietta codice e si forza il salto che lo esegue

## ■ Heap Overflow

- Si sfruttano vulnerabilità specifiche dei metadati inseriti dalle librerie C per l'allocazione dinamica di memoria

## ■ Return Oriented Programming

- Si assembla il codice da eseguire come sequenza di “gadget” (brevi sequenze di linguaggio macchina prese dai binari già in memoria)





# RET2LIBC / RET2SYSCALL

- **Idea di base: non iniettare codice ma usare codice già caricato per altri scopi**
  - funzioni dell'onnipresente standard C library
  - system call
- **Usare stack overflow per iniettare puri dati**
  - riempire lo stack con indirizzi e parametri necessari a eseguire il codice scelto in modo che compia l'azione malevola
  - saltare alla funzione di libreria o attivare la syscall



# RET2LIBC

## ■ Esempio: si vuole eseguire

`system("/bin/sh")`

## ■ Passaggi:

- Trovare l'indirizzo della funzione di libreria `system`
- Trovare un modo di passare sullo stack la stringa `"/bin/sh"`
- Comporre lo stack in modo che alla `ret`, ESP punti alla cella che contiene l'indirizzo di `system` e che questa trovi sullo stack l'indirizzo del parametro atteso (la stringa)
- Si possono collocare strategicamente più indirizzi in modo che il ritorno da una library call ne scateni un altro (es. funzione `exit` se si vuole garantire una terminazione pulita del processo per mostrare un comportamento non anomalo)

## ■ Trovare gli indirizzi non è sempre difficile

- codice compilato staticamente → librerie incluse in `.text`
- codice linkato dinamicamente → entry point inclusi in `.text` come stub che caricano e chiamano la funzione a tempo di esecuzione
- disassemblare il binario è lo strumento principale

# ASLR

- Una prima contromisura contro l'iniezione di codice maligno è l'**ASLR: Address Space Layout Randomization**
  - E' una protezione offerta dall'OS
  - Si tratta di rendere casuale l'indirizzo di partenza dei segmenti che compongono un processo
    - non tutti: indirizzo delle librerie, base dello stack, base dell'heap
  - Questa randomizzazione interessa solo gli indirizzi virtuali di un processo non la sua disposizione in RAM
  - In questo modo l'attaccante non ha modo di trovare un plausibile indirizzo assoluto di ritorno a cui puntare e non può saltare a funzioni di libreria
- Per Linux tale feature è offerta dalla patch grsecurity/PAX che **solo in alcune distribuzioni è inserita nel kernel**



# RET2SYSCALL

- Innescare una syscall invece di saltare a una funzione
  - un po' più contorto
  - serve il codice binario
- Passaggi:
  - Caricare l'identificatore della syscall in EAX
  - Caricare i parametri in EBX, ECX, EDX, ESI, EDI
  - Invocare INT 0x80
- Come eseguire queste operazioni su NX stack?
  - sono semplici e comuni, esisteranno sicuramente pezzi di codice **in .text** che le eseguono e subito o poco dopo incontrano una ret
  - .text non subisce ASLR, indirizzi prevedibili!

```
POP ECX
POP EAX
RET
```

Facili da trovare ovunque una funzione ripristini i registri che ha sporcato

```
POP EBX
RET
```

Facile da trovare ovunque sia legittimamente invocata una syscall

```
INT 0x80
```

- comporre lo stack per concatenarne l'invocazione

Buffer Padding	EBP Padding	Address of code fragment 1	Values to put into registers in an order matching c.f.1	Address of code fragment 2
-------------------	----------------	-------------------------------	--	-------------------------------

# Return-Oriented Programming (ROP)

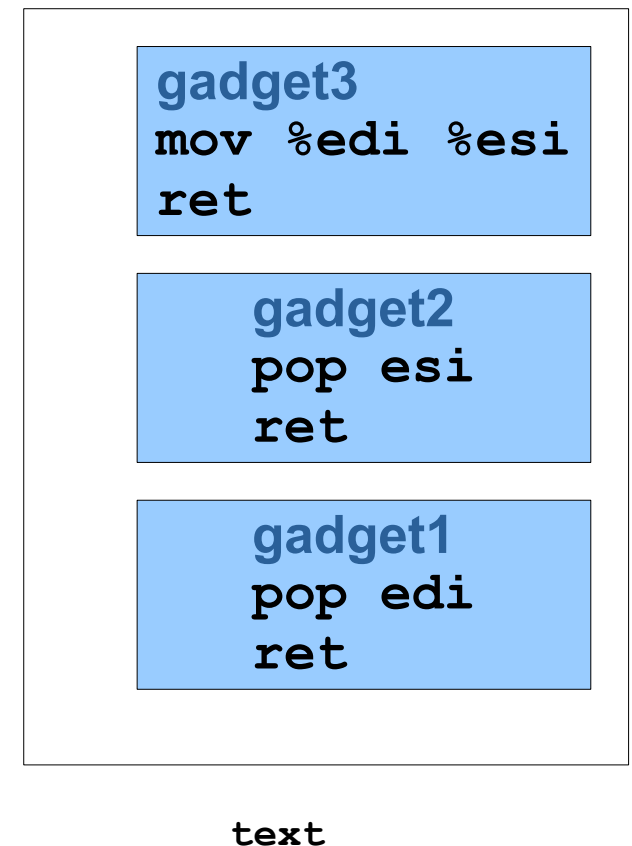
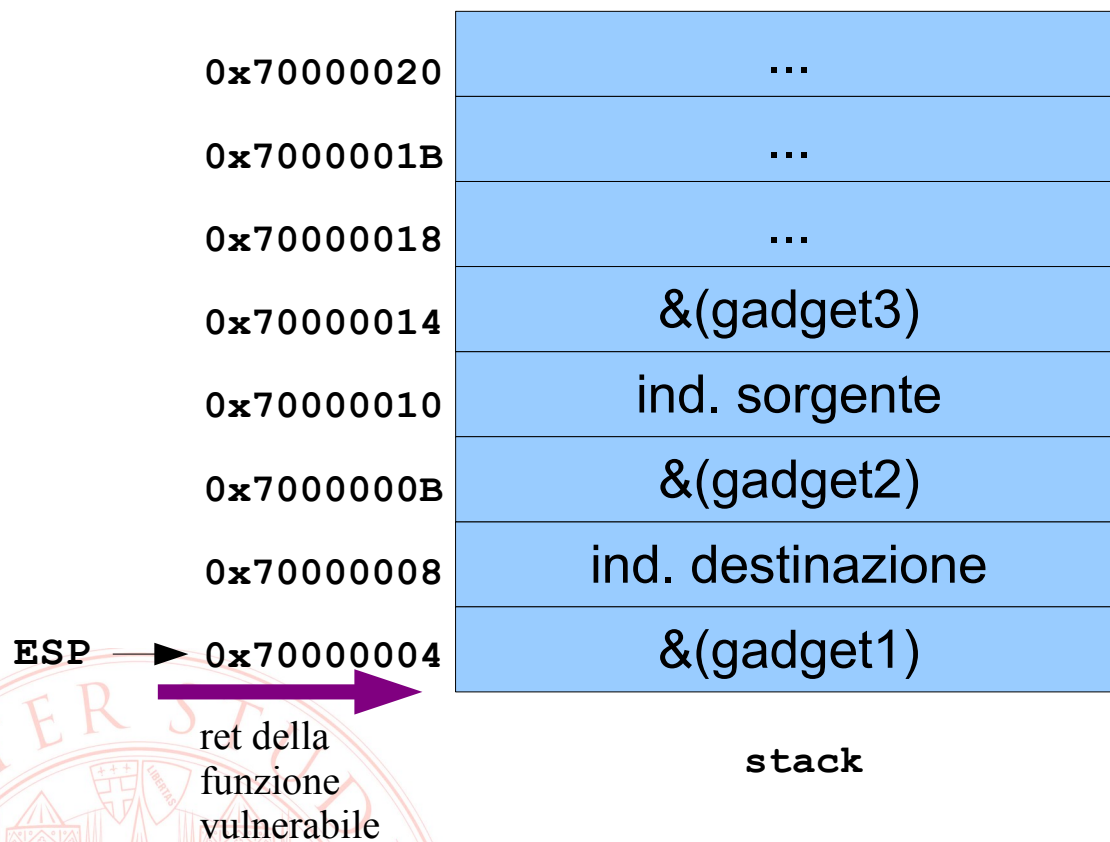
- Il meccanismo può essere generalizzato
- Setacciamo .text cercando qualsiasi **gadget** = sequenza che termini con RET
  - si può sfruttare l'arbitrarietà dell'allineamento



- Iniettiamo la sequenza di indirizzi e parametri sullo stack
  - ogni gadget esegue codice, consuma parametri, e invoca RET
  - RET prende l'indirizzo di "ritorno" dallo stack → salta al gadget successivo

# Return-Oriented Programming (ROP)

## ■ Esempio

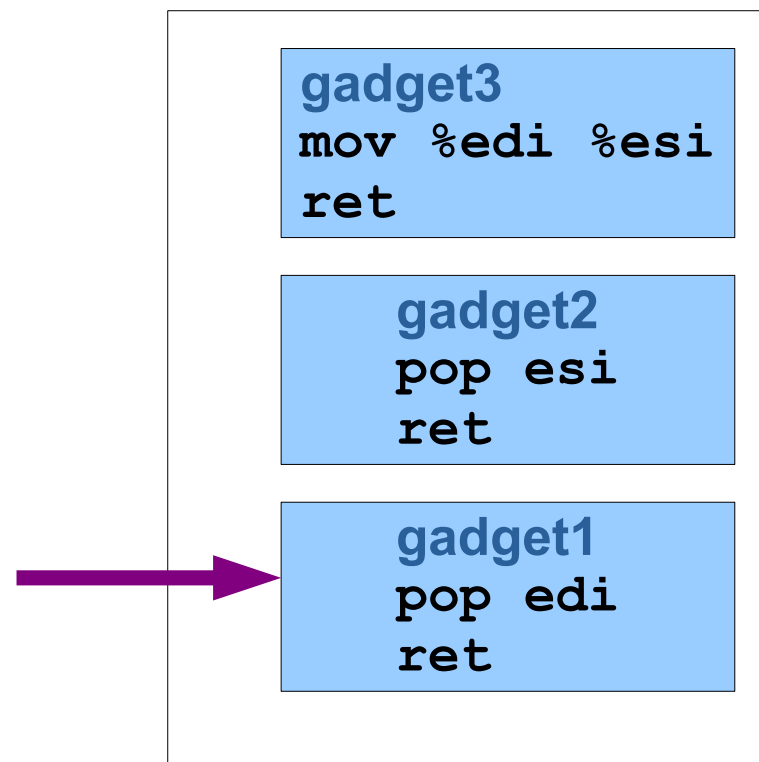


# Return-Oriented Programming (ROP)

## ■ Esempio

0x70000020	...
0x7000001B	...
0x70000018	...
0x70000014	&(gadget3)
0x70000010	ind. sorgente
0x7000000B	&(gadget2)
ESP → 0x70000008	ind. destinazione
0x70000004	

stack



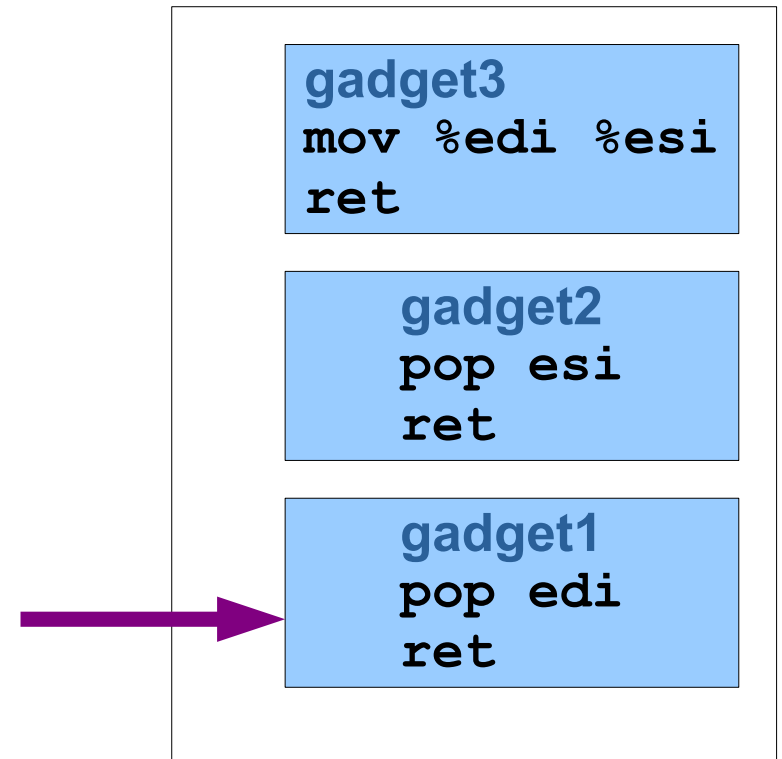
text

# Return-Oriented Programming (ROP)

## ■ Esempio

0x70000020	...
0x7000001B	...
0x70000018	...
0x70000014	&(gadget3)
0x70000010	ind. sorgente
ESP → 0x7000000B	&(gadget2)
0x70000008	
0x70000004	

stack



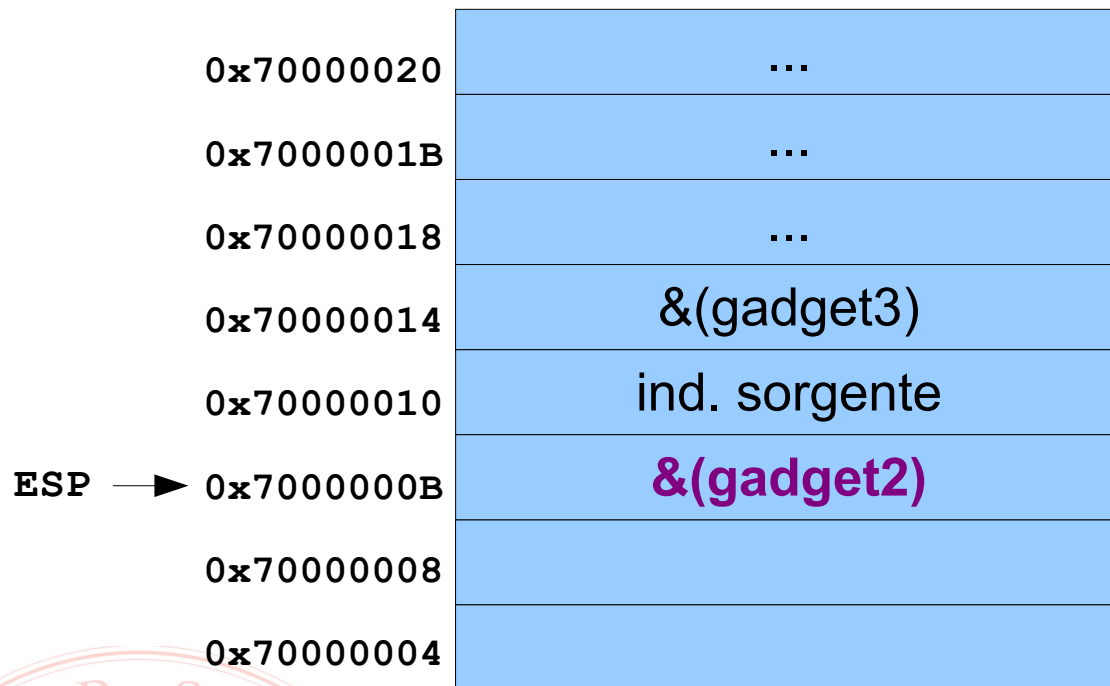
text

edi=ind. destinazione

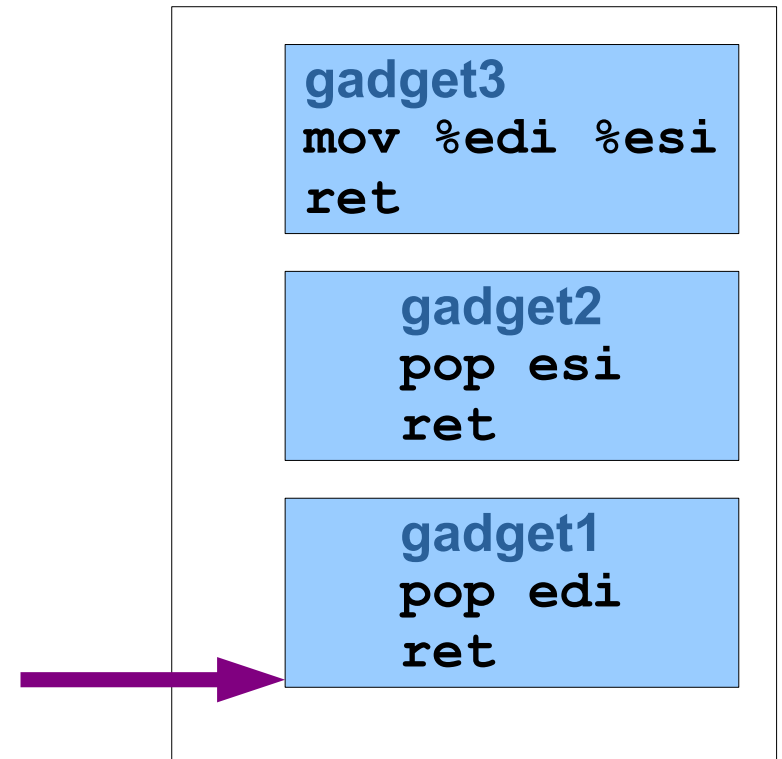


# Return-Oriented Programming (ROP)

## ■ Esempio



stack



text

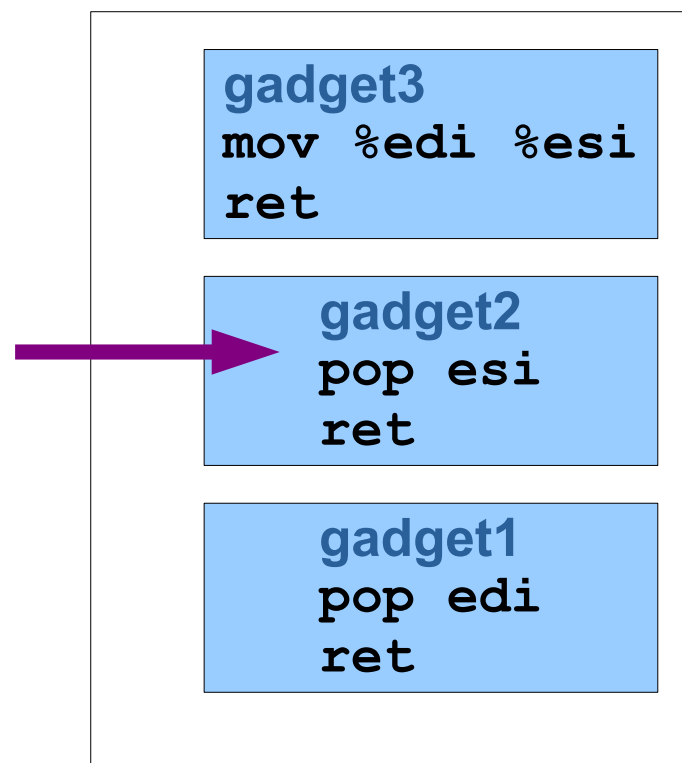
edi=ind. destinazione

# Return-Oriented Programming (ROP)

## ■ Esempio

0x70000020	...
0x7000001B	...
0x70000018	...
0x70000014	&(gadget3)
ESP → 0x70000010	ind. sorgente
0x7000000B	
0x70000008	
0x70000004	

stack



text

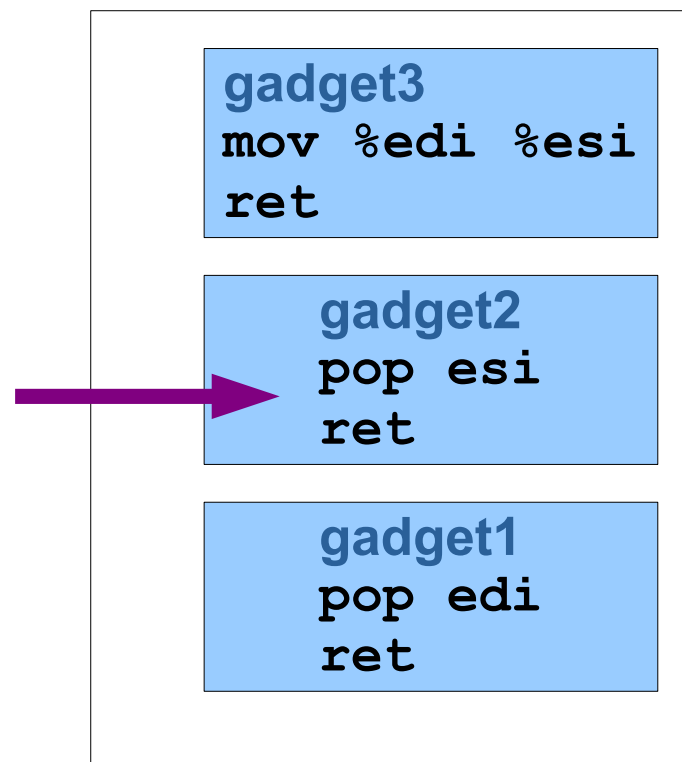
edi=ind. destinazione

# Return-Oriented Programming (ROP)

## ■ Esempio



stack

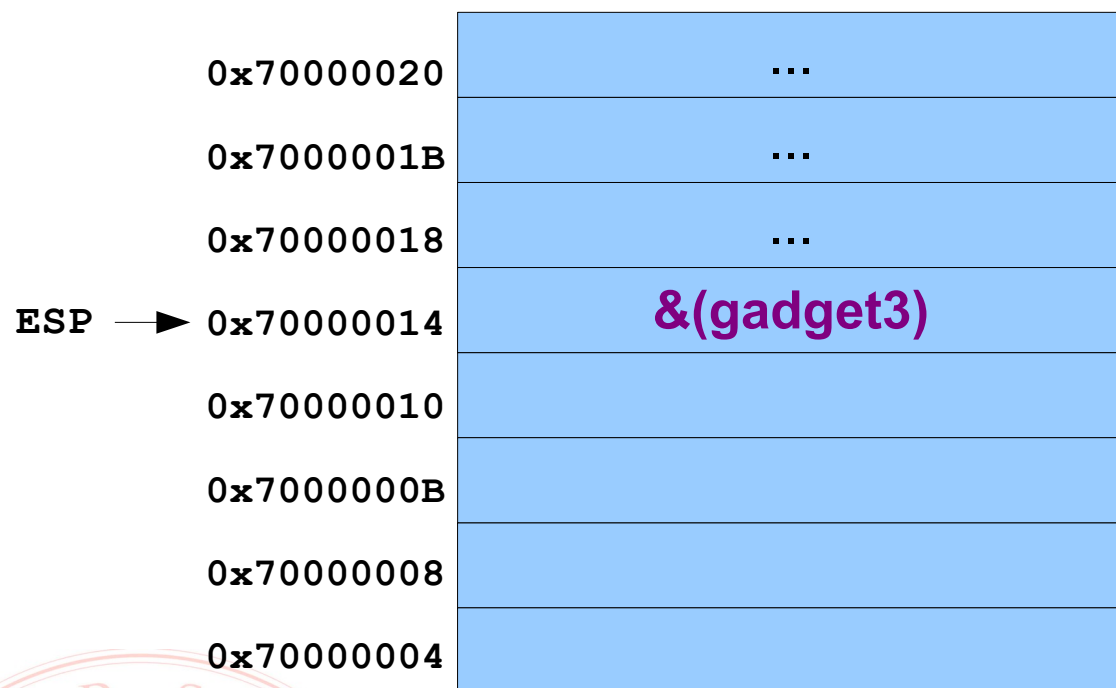


text

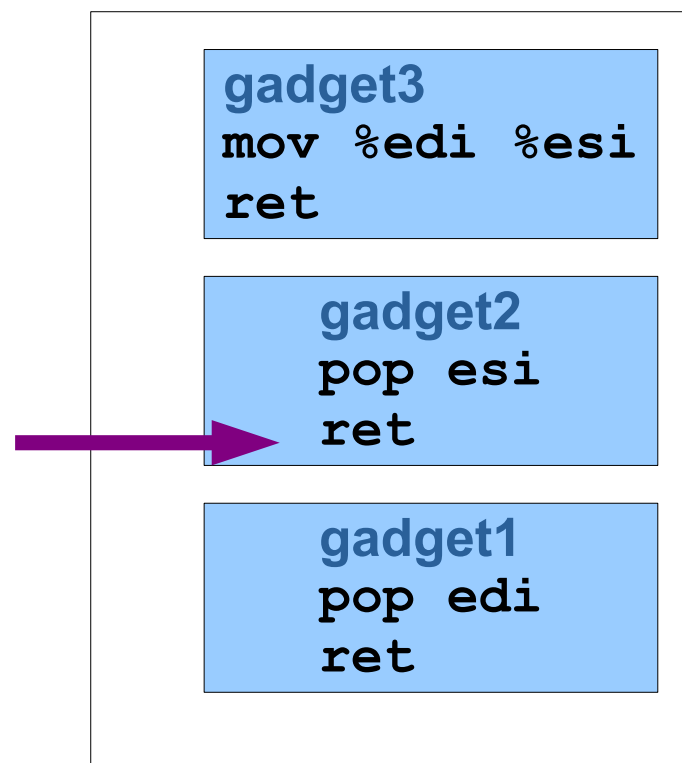
edi=ind. destinazione  
esi=ind. sorgente

# Return-Oriented Programming (ROP)

## ■ Esempio



stack

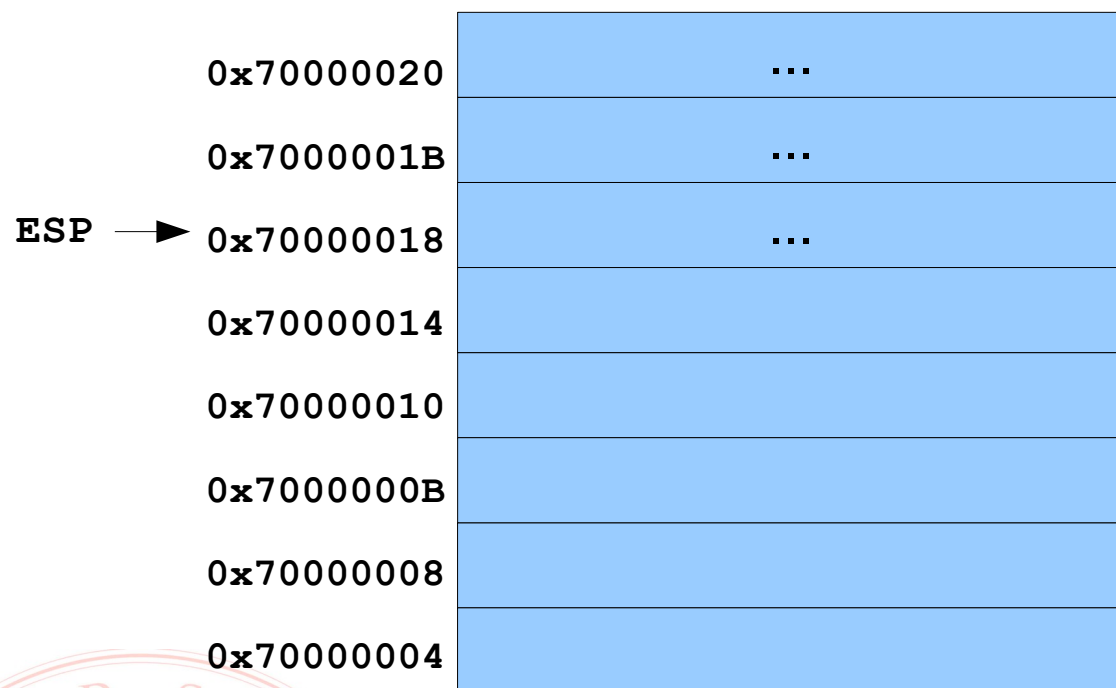


text

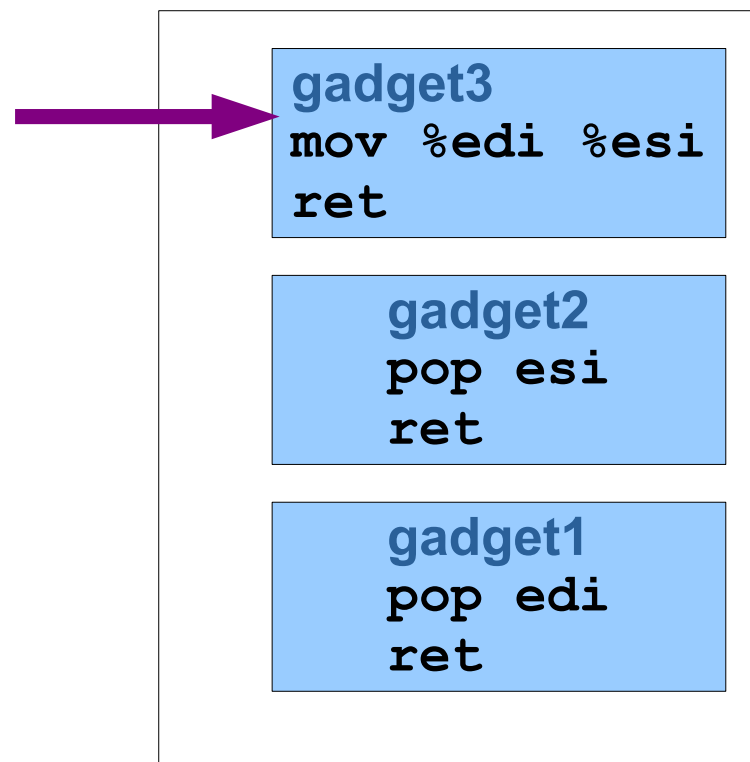
edi=ind. destinazione  
esi=ind. sorgente

# Return-Oriented Programming (ROP)

## ■ Esempio



stack

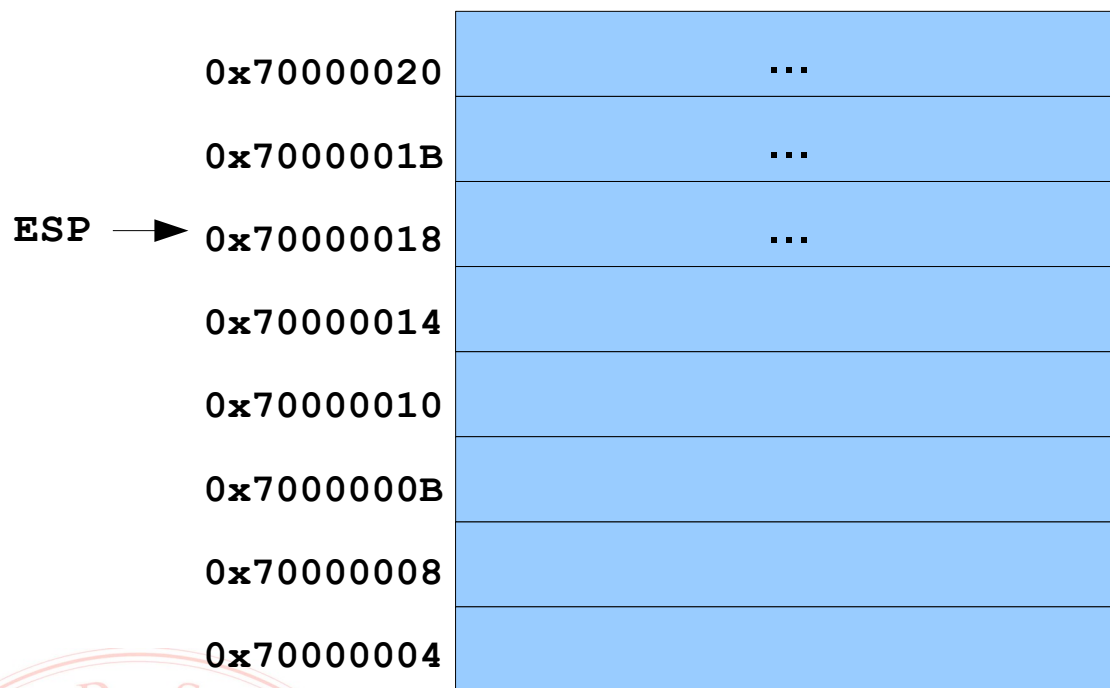


text

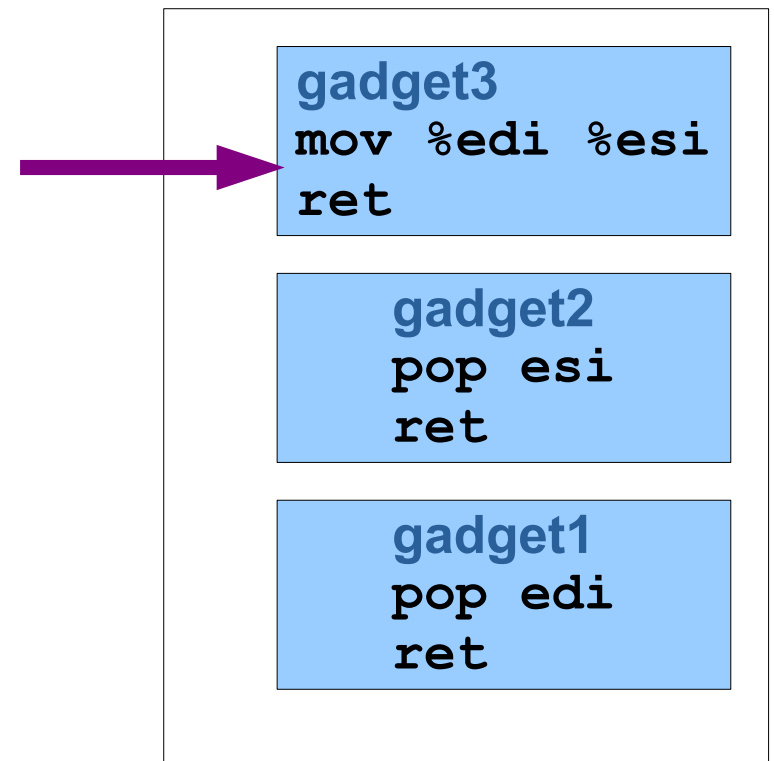
edi=ind. destinazione  
esi=ind. sorgente

# Return-Oriented Programming (ROP)

## ■ Esempio



stack



text

edi=ind. destinazione

esi=ind. sorgente

**dati copiati da sorgente a destinazione!**

# Contromisure parziali e ulteriori estensioni

- NX-Stack e W<sup>X</sup> quasi inutili
  - vari attacchi fanno uso "legittimo" dello stack
- ASLR incompleto
  - non si applica a .text
  - brute forcing degli indirizzi (difficile nelle arch. 64bit)
  - aggirabile utilizzando gli stessi puntatori alle funzioni del codice lecito
  - **estensione: PIE (Position Independent Executable)**
    - .text randomizzato
    - richiede doppia indirazione dei puntatori a funzione (tabelle PLT e GOT)
      - sovrascrivibili per dirottare le invocazioni!
      - **hardening: RelRO**
- Canarini: validi, ma
  - rallentano
  - esistono alcuni modi di aggirarli
    - es. bruteforcing di processi che forkano figli
      - il canarino verrà copiato identico nello stack del figlio
      - provo ad attaccare il figlio, se sbaglio crash
      - faccio generare un altro figlio, che avrà canarino identico
  - **generalizzazione: CFI (Control Flow Integrity)**

# Control Flow Integrity

- "A large **family of techniques** that aims to eradicate memory error exploitation by ensuring that the instruction pointer (IP) of a running process cannot be controlled by a malicious attacker"

<https://www.mdpi.com/2076-3417/9/20/4229/htm>

- Almeno 14 differenti implementazioni note

- Una tipologia rilevante: supporto hardware per puntatori cifrati

- il processo sceglie una chiave di cifratura all'avvio e istruisce la CPU a cifrare e decifrare trasparentemente tutti gli indirizzi di ingresso alle / ritorno dalle funzioni
- un programma vulnerabile può essere DoS-ato ma non dirottato
- esempio: Pointer Authentication Codes sui processori Apple  $\geq$ A12

<https://support.apple.com/en-hk/guide/security/seca5759bf02/web>

[https://www.usenix.org/system/files/sec19fall\\_liljestrand\\_prepub.pdf](https://www.usenix.org/system/files/sec19fall_liljestrand_prepub.pdf)

<https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentic>

